Relit: Implementing Typed Literal Macros in Reason

CHARLES CHAMBERLAIN, University of Chicago
CYRUS OMAR, University of Chicago

Reason is an increasingly popular alternative syntax for OCaml designed to make OCaml more syntactically familiar to contemporary programmers. However, both Reason and OCaml build in literal notation for only a select few data structures, e.g. lists, arrays and, in the case of Reason, a variant on HTML notation. This is unsatisfying because there are many other notations that are familiar to programmers in various domains where OCaml is otherwise well-suited.

In a paper to appear at ICFP 2018, Omar and Aldrich address this deficiency by introducing *typed literal macros (TLMs)*. TLMs allow library providers to define new literal notation for the data structures that they have defined. Unlike prior approaches, e.g. `camlp4` and `ppx`-based string rewriting, both explored in the OCaml ecosystem, TLMs come equipped with powerful abstract reasoning principles — clients do not need to peek at the underlying expansion or the implementation of the parser to reason about types and binding. The paper by Omar and Aldrich investigates these abstract reasoning principles in formal detail.

The proposed talk will provide additional details of Relit, which is our implementation of TLMs for Reason. Relit avoids the need to modify the OCaml compiler itself by making sophisticated use of the existing `ppx` system in OCaml together with an encoding technique based on singleton signatures. We make only a small number of conservative changes to the Reason grammar (and speculate on analagous changes that could be made to the base OCaml grammar to support TLMs in programs written using the OCaml syntax). We reflect on some of the challenges that we faced in interfacing with various components of the OCaml system.

1 MOTIVATION

The Reason project (https://reasonml.github.io/) is experiencing growing adoption by focusing on increasing the *syntactic familiarity* of OCaml (from the perspective of the broader developer community) without changing its semantics. The focus of the Reason effort so far has been on the syntax of the primitive constructs of OCaml. An additional way to increase syntactic familiarity is by introducing new literal forms for constructing and pattern matching on common user-defined data structures. For example, both OCaml and Reason include literal forms for lists, e.g. `[e1, e2, e3]`. By comparison, explicit constructors, e.g. `Nil` and `Cons`, is less concise and less familiar to humans.

The problem is that there are an unbounded number of other user-defined data structures that could benefit from literal notation. Although Reason does include a few more literal notations than OCaml, e.g. it supports "JSX literals" which are based on HTML, it would be infeasible for Reason to attempt to build in all possibly-useful literal notations *a priori*. Instead, there has been a need for a reasonable mechanism that allows ordinary library providers to define new literal notation.

For example, consider the recursive datatype `Regex.t`, defined in Fig. 1a, which encodes simple regular expressions (regexes). Although this encoding is semantically useful, the induced notation is syntactically verbose and unfamiliar. For example, we would construct a regex that matches the strings `"A"`, `"T"`, `"G"` or `"C"`, which represent the four bases in DNA, as follows:

```
module DNA = { let any = Regex.(Or(S "A", Or(S "T", Or(S "G", Or(S "C"))))) }
```

From there, we might define a regular expression that matches the DNA sequences recognized by the BisA restriction enzyme—GCXGC, where X is any of these four bases—as follows:

```
  let bisA = Regex.(Seq(S "G", Seq(S "C",
                    Seq(DNA.any_base, Seq(S "G", S "C")))))
```

Authors' addresses: Charles Chamberlain, University of Chicago, cchamberlain@uchicago.edu; Cyrus Omar, University of Chicago, comar@cs.uchicago.edu.

```
module Regex = {
  type t =
    | Empty
    | AnyChar
    | S(string)
    | Seq(t, t)
    | Or(t, t)
    | Star(t);
}
```

(a) The Regex module, which defines the recursive datatype Regex.t.

```
module RegexNotation = {
  notation $regex at Regex.t {
    lexer  RegexLexer
    parser RegexParser.start
    in package regex_parser;
    dependencies =
      { module Regex = Regex; }
  }
}
```

(b) The definition of the $regex TLM (the lexer and parser is detailed in [Omar and Aldrich 2018]).

```
notation $regex = RegexNotation.$regex; /* or open RegexNotation */
module DNA = { let any_base = $regex `(A|T|G|C)`; };
let bisA = $regex `(GC$(DNA.any_base)GC)`;
```

(c) Examples of the $regex TLM being applied in a bioinformatics application.

Fig. 1. Case Study (reproduced from [Omar and Aldrich 2018]): POSIX-style regex literal notation, with support for regex splicing.

These examples would be more concise and familiar if we built regex literals into Reason, based on the POSIX standard regular expression notation extended with support for constructing regexes compositionally by splicing in expressions of type `Regex.t` delimited by `$(` and `)`:

```
module DNA = { let any_base = `(A|T|G|C)` };
let bisA = `(GC$(DNA.any_base)GC)`
```

However, building in regex notation is *ad hoc* because there are dozens of other examples of notation in programming, mathematics and science that could similarly benefit.

An existing alternative is to use a tool like `camlp4` or `ppx` to introduce this sort of literal notation modularly. However, there are several problems with these tools. To address these problems, Omar and Aldrich [2018] introduced *typed literal macros (TLMs)*, which, unlike these prior tools, allow programmers to reason abstractly—without looking at the underlying expansion or the parser—about (1) which extension is uniquely responsible for each form, (2) the type of each literal, and each spliced expression within the literal, and (3) where variables are bound (i.e. TLMs are hygienic).

An example of a TLM definition for regex notation, named `$regex`, is given in Fig. 1b. We apply the `$regex` TLM to two different literal forms express the example just given in Fig. 1c. Each TLM is responsible for parsing the literal body to which it is applied to generate an expansion, i.e. an OCaml parse tree. The system validates each generated expansion to provide the strong abstract reasoning principles just mentioned. These reasoning principles are examined in formal detail in the paper by Omar and Aldrich [2018], which will be presented by Cyrus at ICFP. The purpose of the proposed talk, which will be given by Charles, will be to provide a complementary "deep dive" that describes how we integrated this mechanism for Reason without modifying the OCaml compiler itself, by making creative use of several existing components of the OCaml system, including the `ppx` preprocessor system, the `compiler-libs` package, `ocamlfind`, `ocamldep` and the module system. These implementation aspects are detailed only superficially in the ICFP paper.

2 IMPLEMENTATION OVERVIEW

We have modified the Reason parser to support the syntax for defining a new notation, seen in Fig. 1b, and the TLM quotation syntax, either `` Path.To.$regex `(body)` `` or `` `(body)` `` (which is used for implicit TLM application, see [Omar and Aldrich 2018]). The literal body can have matching occurrences of `` `( `` and `` )` `` (much like comments in Reason/OCaml), but is otherwise unconstrained.

The `$regex` TLM definition in Fig. 1b desugars to the following OCaml module:

```
module RelitInternalDefn_regex = struct
  type t = Regex.t (* corresponds to "at Regex.t" *)
  module Lexer_RegexLexer = struct end (* ... "lexer RegexLexer" *)
  module Parser_RegexParser = struct end (* ... "parser RegexParser" *)
  module Nonterminal_start = struct end (* ... ".start" *)
  module Package_regex_parser = struct end (* ... "in package regex_parser" *)
  module Dependencies = struct (* ... "dependencies = {" *)
    module Regex = Regex
  end
  exception Apply of string * string (* used internally, see below *)
end
```

The name of the TLM is stored as the module name, with the internal prefix `RelitInternalDefn_`. By encoding TLM definitions as modules, TLMs can be packaged inside other modules (`RegexNotation` in Fig. 1b) and it is straightforward to support scoped TLM abbreviations, as on Line 1 of Fig. 1c. By encoding the relevant information from the TLM definition in module names, e.g. `Lexer_RegexLexer` and `Nonterminal_start`, we ensure that the corresponding signature is a *singleton signature*, i.e. it uniquely identifies the TLM definition, so TLM definitions can be included in module signatures, exported from functors and packaged using the standard tooling.

A TLM application is represented internally as follows:

```
raise (RelitInternalDefn_regex.Apply
        ("You're using relit syntax without the relit ppx!", "A|T|G|C") [@relit])
```

In order to expand the literal body, here `"A|T|G|C"`, we must be able to resolve the signature of the `RelitInternalDefn_regex` module. In other words, we must be able to typecheck before expanding literal bodies. This is why we encode a TLM application as raising an exception. Raised exceptions match against any type, so it will typecheck without an issue before it is expanded fully. Note that this occasionally results in a less specific type error message than the user would otherwise see.

Once the Relit PPX has all the information for a given TLM application site that it needs, it must run the lexer and parser on the literal body (still at compile-time). The lexer and parser must be generated by, or have the same interface as the modules generated by, `ocamllex` and Menhir, respectively. Our first attempt involved starting an OCaml toplevel within the PPX, `#load`-ing the appropriate OCaml object files, and then evaluating a call to the parser. This proved difficult when the parser and lexer themselves had dependencies. Instead, we require that the parser and lexer be in their own `ocamlfind` package, specified in the **`notation`** syntax (here, `regex_parser`). We then call into `ocamlfind ocamlc` internally to ensure that the dependencies are correctly loaded.

After running the provided parser against the literal body, we are left with a *proto-expansion*—an OCaml parse tree where splices are represented abstractly, i.e. they have not yet been expanded. Before expanding the splices, we first check to make sure the proto-expansion is well-typed and context independent: it must not refer to variables, types, or modules from the application site's typing context. Instead, only the types and modules explicitly mentioned in the **`dependencies`** of the TLM, plus the OCaml pervasives, are available to the proto-expansion. In principle, it should be straightforward to enforce this constraint by typechecking against an empty context extended with the dependencies. However, the OCaml `compiler-libs` package does not directly support this functionality because adding the dependencies to the empty typing context exposes the global modules of all packages that the dependencies transitively depend on. Consequently, we first typecheck against this overly permissive typing context, which ensures, at least, that there are no dependencies on non-global bindings. Then, we write the parse tree to a temporary file, and run `ocamldep -modules` against the file. This gives us the list of free modules, which we check explicitly against the dependencies, resulting in a completely context independent proto-expansion.

We can then consider splices. Internally, an encoded splice that says characters 2 through 8 of the literal body should be parsed as an unexpanded expression of type **`string`** looks like this:

```
raise (ignore (2, 8) ; Failure "RelitInternal__Spliced") : string
```

Notice again that we used an exception to ensure that the typechecker can run, as just described.

The PPX collects all the splices in a proto-expansion, checks that the segments are non-overlapping, and parses the corresponding subsequence of the literal body using our extended Reason parser. One might think that these resulting splice parse trees could then replace the splice reference in the proto-expression, but that would allow the TLM to introduce bindings into the splices. Instead, we want to maintain a capture avoidance property, so we replace each encoded splice with a fresh variable, construct a lambda that abstracts over those variables, and immediately apply this lambda to the corresponding splice parse trees. The resulting expansion is of the following form:

```
((fun a b c -> (* ... proto-expansion with variables instead of splices ... *))
  splice1 splice2 splice3)
```

The last step is to repeat the entire process so as to recursively expand any TLM applications exposed by this first phase of expansion, stopping if there are no remaining TLM applications.

3 CONCLUSION

By encoding TLM definitions as modules with singleton signatures, and TLM applications and spliced expressions using exceptions, we have been able to implement a powerful macro system for Reason using the OCaml PPX system and other existing tooling, rather than by extending the OCaml compiler itself (as in previous proposals for OCaml macros [Yallop and White 2015]). TLM definitions follow the same scoping structure as modules. Our evolving implementation is available from https://github.com/cyrus-/relit (by the time of the workshop, we plan to have an initial beta available with improved error reporting and more conveniences, examples and documentation).

We think TLMs would be useful when using "normal" OCaml syntax too. The main issue here is that every TLM must then be able to support both syntaxes within splices. To achieve this, we plan to use Menhir's support for parameterized parsers to pass in syntax-agnostic versions of the helper functions that support splicing.

There are also several remaining questions that have to do with integration with other components of OCaml's tooling. Although Merlin has basic support for TLMs, we need to add support for highlighting spliced segments differently, and we would like for the programmer to be able to request the type of the spliced segment under the cursor. Another difficulty has to do with `ocamldep`, which cannot see inside spliced expressions in a context-free manner to extract dependencies. We plan to use the recently added support for `ppx` extensions in `ocamldep` to eliminate this restriction, but this is non-trivial due to the need to be able to typecheck the unexpanded program. Some build systems, notably `jbuilder`, currently limit `ppx` extensions from being able to access compiler flags in a manner that causes problems for Relit. We currently only test with `rebuild/ocamlbuild`.

Overall, we believe that our experiences developing Relit, which in many ways "stress tested" the PPX system, will be of interest to others working to improve OCaml's tooling, and we believe that Relit itself is an interesting new mechanism of interest to the OCaml community.

REFERENCES

Cyrus Omar and Jonathan Aldrich. 2018. Reasonably Programmable Literal Notation. *To appear, PACMPL* Issue ICFP (2018). https://github.com/cyrus-/ptsms-paper/raw/master/icfp18/syntax-icfp18.pdf

Jeremy Yallop and Leo White. 2015. Modular Macros. *OCaml Users and Developers Workshop* (2015).