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 purpose of this proposed talk is to 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 achieving significant and growing adoption by focusing on the improving the *syntactic familiarity* of OCaml, from the perspective of the broader developer community, without changing its semantics. The focus of this effort has been on the syntax of the primitive constructs of OCaml. However, another way to increase syntactic familiarity is by introducing new literal forms for common user-defined data structures. For example, both OCaml and Reason include literal forms for lists, e.g. `[e1, e2, e3]` in Reason, which lay out the list elements in the familiar sequential, punctuation-separated format. By comparison, explicitly applying constructors, e.g. `Nil` and `Cons`, is both more verbose and less familiar to humans.

The problem is that this general design pattern is not limited to a select few data structures like lists. In fact, there are an unbounded number of 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 is a need for a mechanism that allows ordinary library providers to define new literal notation for their own data structures.

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_base = Regex.(Or(Str "A", Or(Str "T", Or(Str "G", Or(Str "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
    | Str(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 and
    parser RegexParser.start
    in regex_parser;
    dependencies =
      { module Regex = Regex; }
  }
}
```

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

```
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 [?]): POSIX-style regex literal notation, with support for regex splicing.

and 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(Str "G", Seq(Str "C",
  Seq(DNA.any_base,
  Seq(Str "G", Str "C")))))
```

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 values of type Regex.t, these examples would be substantially more concise and familiar to programmers:

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

However, building regex notation for this particular encoding of regexes is, we argue, rather *ad hoc* — there are dozens of other examples of notation in programming, mathematics and science that could similarly benefit [???].

An alternative is to use a tool like camlp4 or ppx to introduce this sort of literal notation modularly. There are several problems with these tools, however. Omar et al. [?] details the problems, but to briefly summarize, these systems lack the ability to reason abstractly—without looking at the underlying expansion itself, or the parsing and expansion logic—about responsibility (which extension is uniquely responsible for each form?), typing (what type does a new form have?) and binding (where are variables bound, given that the expansion is not visible?)

To address this problem, Omar et al. [?] introduced *typed literal macros (TLMs)*. An example of a TLM definition for regex notation, named $regex, is given in Fig. 1b. We can apply this TLM

Motivation similar to that in the ICFP paper – but compare more explicitly to how people use ppx to rewrite string literals to support notation, camlp4, and Reason's built in HTMLish notation.

2 IMPLEMENTATION

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 `(body)` or Path.To.$regex `(body)`.

We parse the $regex notation in Fig. 1b into an OCaml Parsetree equivalent to the following OCaml module:

```
module RelitInternalDefn_regex = struct
  type t = Regex.t
  module Lexer_RegexParser = struct end
  module Parser_RegexLexer = struct end
  module Package_regex_parser = struct end
  module Nonterminal_literal = struct end
  module Dependencies = struct
    module Regex = Regex
  end
  exception Call of string * string
end
```

The name of the TLM, i.e the `regex` in `$regex`, is stored as the module name, with an internal prefix. This enables TLMs to be aliased, opened, and qualified by their parent modules. A TLM application is represented as a call to **`raise`** within the OCaml `Parsetree` generated by the Reason parser:

```
raise (RelitInternalDefn_regex.Call
        ("You're using relit syntax without the relit ppx!", "a|b") [@relit])
```

This passes the body of the TLM application on to the next stage of compilation — the Relit PPX. It also provides a meaningful error message to any users of a Relit notation that forget to use the Relit PPX. The Relit PPX starts by running the OCaml typechecker. Then it maps over the resulting `Typedtree`, searching for all TLM applications. Every node of this typed tree has a typing environment which contains the modules, variables, record fields in scope, and their types or signatures. So to resolve the TLM definition for a given application, the PPX looks up the prefixed module name (in this case `RelitInternalDefn_regex`) in the application's typing environment to find its signuatere. This is why we encode the data for the lexer, the parser, their package, and the starting nonterminal as module names; it can be read from the signature of the notation's module without having access to the module's definition itself.

This method of finding the module through the typing environment results in a strange paradox: We must typecheck before we've expanded the TLM application, even before we have access to its expected type. This is where the **`raise`** call becomes important. Raising an exception matches against every type, so it will typecheck without an issue before it is expanded fully.

(1) Extending the Reason parser
(2) Using singleton signatures to encode definitions
(3) Using exceptions for application + typechecking inside a PPX (but really we only need to signature check but thats not possible). Is it bad?
(4) Using ocamldep to determine module dependencies
(5) Awkwardness with packaging the parser
(6) Speculation: issues with OCaml + Reason support together (use menhir's parameterization?)

3 DISCUSSION

Successes:

(1) ocamldep was a straightforward way to do context independence
(2) can do typechecking and compile-time code execution in a ppx

Challenges:

(1) packaging / loading parsers at compile-time is a huge hassle
(2) jbuilder treats ppx differently
(3) ordering of ppx matters

(4) ocamldep can't see into spliced terms – how to fix?
(5) we need to extend merlin and other tools still...
(6) issues with using OCaml + Reason together

briefly: relationship between this stuff and Scheme/Racket/Scala-style macros