Reasonably Programmable Literal Notation

CYRUS OMAR*, University of Chicago
JONATHAN ALDRICH, Carnegie Mellon University

General-purpose programming languages typically define literal notation for only a small number of common data structures, like lists. This is unsatisfying because there are many other data structures for which literal notation might be useful, e.g. finite maps, regular expressions, HTML elements, SQL queries, syntax trees for various languages and chemical structures. There may also be different implementations of each of these data structures behind a common interface that could all benefit from common literal notation. This paper introduces *typed literal macros (TLMs)*, which allow library providers to define new literal notation of nearly arbitrary design at any specified type or parameterized family of types. Compared to existing approaches, TLMs are uniquely *reasonable*. TLM clients can reason abstractly, i.e. without examining grammars or generated expansions, about types and binding. The system only needs to convey to clients, via secondary notation, the inferred *segmentation* of each literal body, which gives the locations and types of spliced subterms. TLM providers can reason modularly about syntactic ambiguity and expansion correctness according to clear criteria. This paper incorporates TLMs into Reason, an emerging alternative front-end for OCaml, and demonstrates, through several non-trivial case studies, how TLMs integrate with the advanced features of OCaml, including pattern matching and the module system. We also discuss optional integration with MetaOCaml, which allows TLM providers to be more confident about type correctness. Finally, we establish these abstract reasoning principles formally with a detailed type-theoretic account of expression and pattern TLMs for "core ML".

CCS Concepts: • **Software and its engineering → Extensible languages**; **Macro languages**;

ACM Reference Format:
Cyrus Omar and Jonathan Aldrich. 2018. Reasonably Programmable Literal Notation. *Proc. ACM Program. Lang.* 1, 1, Article 1 (September 2018), 32 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

When designing the surface syntax of a general-purpose programming language, it is common practice to define shorthand *literal notation*, i.e. notation that decreases the syntactic cost of constructing and pattern matching over values of some particular data structure or parameterized family of data structures. For example, many languages in the ML family support list literals like `[x1, x2, x3]` in both expression and pattern position [Harper 1997; Milner et al. 1997]. Lists are common across problem domains, but other literal notation is more specialized. For instance, Ur/Web extends the surface syntax of Ur (an ML-like language [Chlipala 2010]) with expression and pattern literals for encodings of XML and HTML data [Chlipala 2015]. For example, Fig. 1 shows two Ur/Web HTML literals, one that "splices in" a string expression delimited by `{[` and `]}` and the other an HTML expression delimited by `{` and `}`.

*The majority of this research was performed while the first author attended Carnegie Mellon University.

Authors' addresses: Cyrus Omar, University of Chicago, Chicago, IL, comar@cs.uchicago.edu; Jonathan Aldrich, Carnegie Mellon University, Pittsburgh, PA, jonathan.aldrich@cs.cmu.edu.

2475-1421/2018/9-ART1
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
fun heading first_name = <xml><h1>Hello, {[first_name]}!</h1></xml>
val body = <xml><body>{heading "World"} ...</body></xml>
```

Fig. 1. HTML literals with support for splicing at two types are built primitively into Ur/Web [Chlipala 2015].

This design practice, where the language designer privileges certain library constructs with built-in literal notation, is *ad hoc* in that it is easy to come up with other examples of data structures for which mathematicians, scientists or programmers have invented specialized notation [Cajori 1928; Iverson 1980; Omar et al. 2014]. For example, (1) clients of a "collections" library might want not just list literals, but also literal notation for matrices, finite maps, and so on; (2) clients of a "web programming" library might want CSS literals (which Ur/Web lacks); (3) a compiler author might want "quotation" literals for the terms of the object language and various intermediate languages of interest; and (4) clients of a "chemistry" library might want chemical structure literals based on the SMILES standard [Anderson et al. 1987].

Although requests for specialized literal notation are easy to dismiss as superficial, the reality is that literal notation, or the absence thereof, can have a substantial influence on software quality. For example, Bravenboer et al. [2007] finds that literal notation for structured encodings of queries, like the SQL-based query literals now found in many languages [Meijer et al. 2006], reduce the temptation to use string encodings of queries and therefore reduce the risk of catastrophic string injection attacks [OWASP 2017]. More generally, evidence suggests that programmers frequently resort to "stringly-typed programming", i.e. they choose strings instead of composite data structures, largely for reasons of notational convenience. In particular, Omar et al. [2014] sampled strings from open source projects and found that at least 15% of them could be parsed by some readily apparent type-specific grammar, e.g. for URLs, paths, regular expressions and many others. Literal notation, with support for splicing, would decrease the syntactic cost of composite encodings, which are more amenable to programmatic manipulation and compositional reasoning than string encodings.

Of course, it would not scale to ask general-purpose language designers to build in support for all known notations *a priori*. Instead, there has been persistent interest in mechanisms that allow library providers to define new literal notation on their own. For example, direct grammar extension systems like Camlp4 [de Rauglaudre 2003] and Sugar* [Erdweg et al. 2011; Erdweg and Rieger 2013], term rewriting systems like Template Haskell [Mainland 2007; Sheard and Peyton Jones 2002], the system of *type-specific languages* (TSLs) described by Omar et al. [2014], and other systems that we will discuss below can all be used to define new literal notation (and, in some cases, other forms of new notation, such as new infix operator forms, control flow operations or type declaration forms, which we leave beyond the scope of this paper).

Problem The problem that specifically motivates this paper is that these existing systems make it difficult or impossible to reason abstractly about such fundamental issues as types and variable binding when presented with a program using user-defined literal forms. Instead, programmers and editor services can only reason transparently, i.e. by inspecting the underlying expansion or the implementation details of the collection of extensions responsible for producing the expansion.

Consider, for instance, the perspective of a programmer trying to comprehend the program text in Fig. 2a, which is written in an emerging dialect of OCaml's surface syntax called Reason [Reason Team 2018] that has, hypothetically, been extended with some number of new literal forms by a grammar extension system — Lines 1-6 outline the Camlp4 mechanism [de Rauglaudre 2003]; Sugar*/SugarHaskell is similar [Erdweg et al. 2011, 2012]. Line 8 uses one of these active syntax extensions to construct an encoding of a query in the rather obscure database query language K, using its intentionally terse notation [Whitney and Shasha 2001]. The problem is that a programmer examining the program as presented, and unfamiliar with (i.e. holding abstract) the details elided on Lines 1-6, cannot easily answer questions like the following:

EXISTING GRAMMAR EXTENSION SYSTEMS

```
1 EXTEND /* loaded by, e.g., camlp4 */
2   expr:
3     |  "`(" q = kquery ")`" -> q
4   kquery:
5     | /* ...K query grammar... */
6 /* ...more extensions defined... */
7 let x = compute_x();
8 let y = `((!R)@&{&/x!/:2_!x}'!R)`;
```

(a) It is difficult to reason abstractly given program text that uses a variety of grammar extensions (see the six reasoning criteria in the paper text).

THIS PAPER: TYPED LITERAL MACROS

```
1 notation $kq at KQuery.t {
2   lexer  KQueryLexer
3   parser KQueryParser.start
4   in package kquery_parser;
5   dependencies = {module KQuery = KQuery}
6 }; /* ...more notations defined... */
7 let x = compute_x();
8 let y = $kq `((!R)@&{&/x!/:2_!x}'!R)`;
```

(b) TLMs make examples like these more reasonable by leaving the base grammar fixed and strictly enforcing a simple type, binding and segmentation discipline.

Fig. 2. Two of the possible ways to introduce literal notation for encodings of K queries

(1) **Responsibility**: Which syntax extension determined the expansion of the literal on Line 8? Might activating a new extension generate a conflicting expansion for the same literal?
(2) **Expansion Typing**: What type does the expansion, and thus the variable `y` on Line 8, have?
(3) **Context Dependence**: Which bindings does the expansion of Line 8 invisibly depend on? If we shadow or remove a module or other binding, could that break or change the meaning of Line 8 because its expansion depends invisibly on the original binding?
(4) **Segmentation**: Are the characters `x`, `R` and `2` on Line 8 parsed as spliced expressions, meaning that they appear directly in the underlying expansion, or are they parsed in some other way peculiar to this literal notation, e.g. as operators in the K query language?
(5) **Segment Typing**: What type is each spliced term expected to have? How can we infer a type for a variable that appears in a spliced term without looking at where it ends up in the expansion?
(6) **Capture**: If `x` is in fact a spliced term, does it refer to the binding of `x` on Line 7, or might it capture an invisible binding of the same identifier in the expansion of Line 8?

Forcing the programmer to reason transparently to answer basic questions like these defeats the ultimate purpose of syntactic sugar: decreasing cognitive cost [Green 1989]. analogous problems do not arise when programming without syntax extensions in languages like ML — programmers can reason lexically about where variables and other symbols are bound, and types mediate abstraction over function and module implementations [Reynolds 1983]. Ideally, the programmer would be able to abstract in some analogous manner over the implementation of an unfamiliar notation.

Given these issues, we concluded that direct grammar extension systems like `camlp4` were not ideally suited for integration into the Reason platform, which seeks to develop a clear and modern surface syntax for the OCaml programming language [Reason Team 2018]. We also evaluated various approaches that are based not on direct grammar extension but on term rewriting over a fixed grammar. We give a full account of this evaluation in Sec. 7, but briefly, we found that:

- Unhygienic approaches like OCaml's preprocessor extension point (PPX) rewriters [Leroy et al. 2014] and Template Haskell [Mainland 2007; Sheard and Peyton Jones 2002] allow us to define new literal notation with support for splicing by repurposing existing string literal forms. They also partially or completely solve the problem of reasoning about **Responsibility** but they do not satisfy the remaining five reasoning criteria.
- Hygienic term rewriting macro systems, like those in various Lisp-family languages, e.g. Racket [Flatt 2012], as well as Scala [Burmako 2013], do not allow us to flexibly repurpose string literal forms to define composite literal forms because the hygiene discipline cannot account for base language terms spliced out of string literal bodies via parsing (see Sec. 7 for more details).

- *Type-specific languages (TSLs)* [Omar et al. 2014] come closer to our goals in that they explicitly support splicing terms out of literal bodies, but the mechanism falls subtly short with regard to the six reasoning principles just discussed in ways that we detail in Sec. 7. In any case, this approach was designed for simple nominally-typed languages and relies critically on a particular local type inference scheme. It is not immediately suitable for a language with an ML-like semantics, meaning a language with support for structural types (like tuple and function types in ML), parameterized type families, pattern matching and non-local type inference.

Contributions This paper introduces *typed literal macros* (TLMs): the first system for defining new literal notation that (1) provides the ability to reason abstractly about all six of the topics just outlined; and (2) is semantically expressive enough for integration into Reason/OCaml and other full-scale statically typed functional languages. We evaluate these claims with a number of non-trivial examples appearing throughout the paper that involve the advanced language features mentioned above. In describing these examples, we demonstrate that literal parsing logic can be defined using standard, unmodified parser generators, so the burden on notation providers is comparable to that of existing systems despite these stronger reasoning principles. Finally, we give a type-theoretic account of TLMs where we formally establish these abstract reasoning principles.

A Brief Overview For a brief overview of the proposed mechanism, consider Fig. 2b. Lines 1-6 define a TLM named `$kq` that provides K query literal notation. Line 8 applies this TLM to express the example from Fig. 2a. We can reason abstractly about this program as follows.

(1) **Responsibility**: The lexer and parser specified by the applied TLM on Lines 2-4 are together exclusively responsible for lexing, parsing and expanding the body of the generalized literal form, i.e. the characters between `` `( `` and `` )` ``. We will give more details on generalized literal forms and on constructing a TLM lexer and parser in the next section. For now, let us simply reiterate that our design goal is to provide a system where the programmer does not normally need to look up the definitions of `KQueryLexer` and `KQueryParser` to reason about types and binding.
(2) **Expansion Typing**: The type annotation on Line 1 specifies the type that every expansion generated by `$kq` must have, here `KQuery.t`.
(3) **Context Dependence**: Line 5 specifies that expansions generated by `$kq` are allowed to use the module `KQuery`, and no others. The system ensures that this dependency is bound as specified even if the variable `KQuery` has been shadowed at the application site. This completely relieves clients from needing to consider expansion-internal dependencies when naming variables.
(4) **Segmentation**: The intermediate output that the TLM generates is structured so that the system can infer from it an accurate *segmentation* of the literal body that distinguishes spliced terms, i.e. those that appear in the expansion, from segments parsed in some other way. The segmentation is all that needs to be communicated to language services downstream of the expander, e.g. editors and pretty printers, which can pass it on to the programmer using secondary notation, e.g. colors in this document. So by examining Line 8, the programmer knows that the two instances of `x` are spliced expressions (because they are in black), whereas the `R`'s must be parsed in some other way, e.g. as operators of the K language (because they are in lavender). Errors in spliced terms can always be reported in terms of their original location.
(5) **Segment Typing**: Each spliced segment in the inferred segmentation also has a type annotation. This, together with the context independence condition, ensures that type inference at the TLM application site can be performed (by editor services or in the programmer's mind) abstractly, i.e. by reference only to the type annotations on the spliced segments, not the full expansion.
(6) **Capture**: Splicing is guaranteed to be capture-avoiding, so the spliced expression `x` must refer to the binding of `x` on Line 7. It cannot have captured a coincidental binding of `x` in the expansion.

Paper Outline Sec. 2 details expression TLMs in Reason with several more case studies of varying detail, notably including TLMs for regular expressions, HTML literals, chemical structure literals and quasiquotation for Reason language terms, which can be used for implementing other TLMs. It also describes experimental integration with MetaOCaml, which can help providers reason about type correctness. Sec. 3 then briefly introduces pattern TLMs and describes the special reasoning conditions in pattern position. Sec. 4 introduces the more general parametric TLMs, which allow us to define literal notation at a type- or module-parameterized family of types. Having introduced the basic machinery by example, we proceed in Sec. 5 to describe a type-theoretic account of simple expression and pattern TLMs and formally establish the reasoning principles implied above in their essential form. The full technical details and proofs are in the accompanying technical report [Omar and Aldrich 2018]. Sec. 6 provides a brief overview of how we are implementing TLMs for Reason without modifying OCaml's type system. This implementation, called `Relit`, and additional implementation details, documentation and examples are available from the `Relit` project page:

https://github.com/cyrus-/relit

Sec. 7 compares TLMs to related work, guided by the rubric of reasoning principles just discussed. Sec. 8 concludes with a discussion of contributions, limitations and future work.

2 EXPRESSION LITERALS

Consider the recursive datatype `Regex.t` defined in Fig. 3a, which encodes regular expressions (regexes) into Reason [Thompson 1968]. Regexes are common in, for example, bioinformatics, where they are used to express patterns in DNA sequences. For example, we can construct a regex that matches the strings `"A"`, `"T"`, `"G"` or `"C"`, which represent the four bases in DNA, as follows:

```
let any_base = Regex.(Or(Str "A", Or(Str "T", Or(Str "G", Or(Str "C")))))
```

Note that in Reason, the notation `Regex.(`e`)` locally opens the module `Regex` within e, so we do not need to qualify each constructor application. Even with this shorthand, however, constructing regexes in this way is syntactically costly. Instead, we would like to have the option to use the common POSIX-style notation [IEEE 2016] when constructing values of type `Regex.t`, including values constructed compositionally from other regexes and strings. We solve this problem in Fig. 3b by defining a TLM named `$regex` (pronounced "lit regex") that supports POSIX-style regex notation extended with splice forms for regexes (delimited by `$(` and `)`) and strings (delimited by `$$(` and `)`).

Fig. 3c shows three examples of `$regex` being applied. Line 2 applies `$regex` to construct the regex `DNA.any_base` that was described above, this time using the more concise and common POSIX regex notation. Line 3 applies `$regex` again, using its regex splice form to compositionally construct a regex matching DNA sequences recognized by the BisA restriction enzyme, where the middle base can be any base. Finally, Lines 4-5 define a function, `restriction_template`, that constructs a more complex regex from these first two regexes and a given gene sequence represented as a string.

2.1 Client Perspective

Let us start from the perspective of a client programmer examining Fig. 3 but holding the underlying expansion of Fig. 3c, as well as the details of the lexer and parser, `RegexLexer` and `RegexParser`, abstract. We will return to describe the lexer and parser from the provider's perspective in Sec. 2.2.

Let us consider the second of these three TLM applications more closely:

```
$regex `(GC$(DNA.any_base)GC)`
```

According to the context-free syntax of (this paper's extension to) Reason, this form is a leaf of the unexpanded parse tree, like a string literal would be. TLM names are prefixed by `$` to distinguish them from variables. We call the TLM argument, `` `(GC$(DNA.any_base)GC)` ``, a *generalized literal form*, following our prior work on type-specific languages [Omar et al. 2014]. The only lexical constraint

```
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`.

```
1 module RegexNotation = {
2   notation $regex at Regex.t {
3     lexer   RegexLexer
4     parser  RegexParser.start
5     in package regex_parser;
6     dependencies = {module Regex = Regex}
7   };
8 }
```

(b) The definition of `$regex`. Fig. 6 defines `RegexLexer` and `RegexParser`, which are detailed in Sec. 2.2.

```
1 notation $regex = RegexNotation.$regex; /* or open RegexNotation */
2 module DNA = { let any_base = $regex `(A|T|G|C)`; };
3 let bisA = $regex `(GC$(DNA.any_base)GC)`;
4 let restriction_template = (gene) =>
5   $regex `($(bisA)$(DNA.any_base)*$$(gene)$(DNA.any_base)*$(bisA))`;
```

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

Fig. 3. Case Study: POSIX-style regex literal notation, with support for string and regex splicing.

imposed on the literal body, i.e. the characters between `` `( `` and `` )` ``, is that any nested occurrences of `` `( `` must be balanced by `` )` ``, much like nested comments in Reason/OCaml. Generalized literal forms therefore lexically subsume many other literal forms. This nesting constraint is to allow TLM applications to appear inside spliced expressions. An example of nested TLM applications is shown in Fig. 5, discussed later in this section. Our prior work [Omar et al. 2014] specified other choices of outer delimitation, including layout-sensitive delimitation, but for this paper `` `( `` and `` )` `` suffice.

2.1.1 Responsibility Responsibility for lexing, parsing and expanding each literal body is delegated uniquely to the applied TLM. TLM definitions and abbreviations (like the abbreviation on Line 1 of Fig. 3c) follow the same scoping rules as Reason modules, i.e. they can appear in modules and be accessed through module paths. When a TLM definition appears inside a module with an explicitly specified module type (a.k.a. signature), it must also appear in the module type with the same specification, up to the usual notions of type and module path equivalence in the type annotation and dependencies, which are discussed below. This is much like the situation with datatype definitions in ML. By convention, we define TLMs in a module suffixed with `Notation` so that client programmers can **`open`** just the relevant TLM definition(s) without bringing other definitions into scope. We will demonstrate a simple lexically scoped implicit application mechanism for situations where the same TLM is being repeatedly applied in Sec. 2.2.2.

What is fundamental about this design is that there is a well-defined protocol that follows the usual scoping rules of the language for finding the definition of the TLM uniquely responsible for each generalized literal form in a program. An editor service or documentation tool could use this protocol to integrate TLM definition lookup into a "go to definition" command. In Sec. 6, we will describe how we use an encoding of TLM definitions as modules with singleton signatures to avoid having to primitively extend OCaml.

2.1.2 Expansion Typing Having found the definition of the `$regex` TLM, the client can immediately determine the type of the expansion being generated at each application site because it is specified explicitly by the clause **`at`** `Regex.t` on Line 2 of Fig. 3b. The expansion type of a TLM is analogous to the return type of a function. The identity of `Regex.t` is determined relative to the TLM definition site, not at the application site, so the module `Regex` need not be in scope at the application site, or it can have been shadowed by a different module.

```
$regex `(GC$(DNA.any_base)GC)`
```

The applied TLM lexes and parses the literal body to generate the **proto-expansion**.

```
Regex.Seq(Regex.Str "GC", Regex.Seq(spliced<4..16 : Regex.t>, Regex.Str "GC"))
```

The system performs proto-expansion validation and recursively expands the splices to generate the final **expansion**.

```
Regex.Seq(Regex.Str "GC", Regex.Seq(DNA.any_base, Regex.Str "GC"))
```

Fig. 4. TLM applications expand in two steps.

It is also worth noting here that although there is no direct mechanism for applying TLMs at the module level, this can be achieved by using OCaml's first class modules [Leroy et al. 2014].

2.1.3 Context Dependence The system enforces a strong context independence condition on generated expansions by requiring that the TLM definition explicitly specify all modules that the expansions it generates might internally depend on. In this case, Line 6 of Fig. 3b specifies that generated expansions might use the module `Regex`, again as it is bound at the TLM definition site, using the module variable `Regex` internally. In general, the dependency can be an arbitrary module path, e.g. `module List = Core.Std.List`. The `Pervasives` module is implicitly opened in each expansion. All other bindings, whether at the TLM application site or the TLM definition site, are not internally available to the expansion.

From the client's perspective, the benefit of this application site context independence discipline is clear—clients do not need to give any thought to which bindings the expansion might invisibly be assuming are in scope, as they do when using unhygienic approaches (see Sec. 7).

The benefits of making the macro definition site dependencies explicit, rather than implicitly allowing expansions to access all bindings at the definition site as in many existing macro systems, arise from the fact that this exposes the TLM's dependencies in the signature of the module where the TLM is defined. This is useful for build tools that extract dependencies by source code analysis. Moreover, if an "internal" module is being used by expansions, then this will be manifest in the signature and can be corrected if this was unintended. Implicit access to the definition site would require the implementation to carefully "smuggle out" otherwise internal values to each application site, thereby skirting the abstraction discipline of ML's module system [Culpepper et al. 2005].

Enforcing this strong context independence condition is technically subtle because TLM parsers need to be able to implement splicing, i.e. they need to be able to parse terms out of the literal body for placement in the expansion. Naïvely checking that only the explicitly named dependencies are free in the expansion would inappropriately constrain application site spliced expressions, which should certainly not be prevented from referring to variables in scope at the application site. For example, consider the bottom of Fig. 4, which shows the final expansion of the example from Line 2 of Fig. 3c. In this term, both `Regex` and `DNA` are free module variables. There is nothing to distinguish references to `DNA` that arose from a spliced sub-expression parsed out of the literal body from those that would indicate that the context independence condition has been violated.

To address this problem, TLM parsers do not generate the final expansion directly, but rather a *proto-expansion* that refers to spliced terms indirectly by location relative to the start of the provided literal body. For example, the proto-expansion generated by `$regex` for the example above can be pretty printed as shown in the middle of Fig. 4. Here, `spliced<4..16 : Regex.t>` is a reference to the spliced expression `DNA.any_base` because the zero-indexed subsequence $[4, 16)$ of `GC$(DNA.any_base)GC` is `DNA.any_base`. We return to the type annotation on the splice reference, `Regex.t`, when we discuss **Segment Typing** in Sec. 2.1.5 below. The context independence condition can be enforced directly on the proto-expansion—the only free variable in the proto-expansion is `Regex`, which is an explicitly listed dependency in Fig. 3b, so all is well.

```
$html `( <div>
           <h3>Chemical Structure of Sucrose</h3>
           <$>$smiles `({mono_glucose}-O-{mono_fructose})` |> Smiles.to_svg</$>
         </div> )`
```

Fig. 5. A practical demonstration of nested TLM application. The colors communicate the segmentation.

2.1.4 Segmentation

The finite set of splice references in the proto-expansion generated for a literal body is called the *segmentation* of that literal body. The segmentation of the example above is the finite set containing one splice reference, `spliced<4..16 : Regex.t>`. For the more complex example from Line 5 of Fig. 3c, the segmentation contains five splice references:

```
{ spliced<2..6 : Regex.t>, spliced<9..21 : Regex.t>, spliced<26..30 : string>,
  spliced<33..45 : Regex.t>, spliced<49..53 : Regex.t> }
```

The system checks that the segmentation does in fact segment the literal body, i.e. that the segments are in-bounds, of positive extent and non-overlapping. Adjacent spliced segments must also be separated by at least one character. The spliced segment locations can therefore be communicated unambiguously to the programmer by tools downstream of the expander, e.g. program editors and pretty printers, using secondary notation. In this paper, non-spliced segments are shown in color and spliced segments start in black.

When TLM applications are nested, a distinct color can be used at each depth. For example, Fig. 5 shows a program fragment where we transform an encoding of a chemical structure expressed using the standard SMILES notation for chemical structures [Anderson et al. 1987], extended with splicing notation, into a vector graphic, then embed this directly into a fragment of a webpage. We will detail the TLM `$html at Html.t`, which implements HTML notation similar to that found in Ur/Web, in Sec. 2.2.5, and we assume a TLM `$smiles at Smiles.t`, and a function, `Smiles.to_svg : Smiles.t => Html.t`, not shown. In Reason, `|>` is reverse function application.

2.1.5 Segment Typing

Each splice reference in the segmentation carries not just the location of a spliced expression but also its expected type. The identity of this type is resolved in a context-independent manner, assuming only the dependencies explicitly specified by the TLM.

By associating a type with each spliced segment, type inference can be performed abstractly, meaning that only the segment types, together with the expansion types specified by the applied TLMs, are necessary to infer types for variables appearing in a client-side function. For example, consider the function `restriction_template` on Lines 4-5 of Fig. 3c. The return type of this function can be inferred to be `Regex.t` from the expansion type annotation on the `$regex` TLM, as previously discussed. The type of the argument, `gene`, can be inferred to be `string` because the segmentation specifies the type `string` for the spliced segment where it appears (cf. the segmentation shown in Sec. 2.1.4 above). The context independence condition implies that `gene` cannot appear elsewhere in the expansion, and so no further typing constraints could possibly be collected from examining the portions of the (proto-)expansion being held abstract. Another important benefit of explicitly tracking the locations of spliced segments is that errors that originate in spliced terms can be reported in terms of their original source location [van Deursen et al. 1993].

Segment types can be communicated directly to the programmer upon request by an editor service For Reason, we plan to equip the Merlin tool [Bour et al. 2018], which is used by various Reason editor extensions (e.g. for Emacs and Vim), with a new editor command that reports the expected type of the innermost spliced segment containing the cursor. Note that because the type is explicitly stated, this information can be reported even when there is a parse or type error in a spliced expression.

Segment types are somewhat analogous to the argument types of a function. The difference is that the argument signature of a function is the same every time the function is applied, i.e. it is associated with the function itself, whereas the segmentation can differ for each choice of literal body. This, of course, is what gives TLMs substantially more notational flexibility, even relative to infix or mixfix function notation (where the number of subexpressions is fixed [Wieland 2009]).

2.1.6 Capture In discussing the question of inferring a type for gene above, we neglected to consider one critical question: are we sure that the variable gene in the third spliced segment on Line 5 of Fig. 3c is, in fact, a reference to the argument gene of the restriction_template function? After all, if we hold the expansion abstract then it may well be that the third spliced segment appears under (i.e. captures) a different binding of the identifier gene. For more common identifiers, e.g. tmp, inadvertent capture is not difficult to imagine. For example, consider this application site:

```
let tmp = /* ... application site temporary ... */;
$html `(<h1><$>f(tmp)</$></h1>)`;
```

Now consider the scenario where the proto-expansion generated by $html has the following form:

```
let tmp = /* ... expansion-internal temporary ... */;
Html.H1Element(tmp, spliced<7..13 : Html.t>);
```

If the final expansion was produced naïvely, by syntactically replacing the splice reference with the final expansion recursively determined for the corresponding spliced expression, then the variable tmp in the spliced expression would capture the expansion-internal binding of tmp. The result if the types of the two bindings differed would be a type error exposing the internal details of the expansion. If the types of the two bindings of tmp coincided, then there would be no static indication of the problem but there could be subtle and mysterious changes in run-time behavior.

To address this problem, splicing is guaranteed to be capture-avoiding. The final expansion is generated by recursively expanding each spliced expression and then inserting it into the final expansion via capture-avoiding substitution, which automatically alpha-varies the internal bindings of the proto-expansion as necessary. There is no need for TLM providers to manually deploy a mechanism that generates fresh variables (as in, e.g., Racket's reader macros [Flatt 2012], further discussed in Sec. 7). For example, the final expansion of the example above is alpha-equivalent to the following:

```
let tmp = /* ... application site temporary ... */;
let tmp_fresh = /* ... expansion-internal temporary ... */;
Html.H1Element(tmp_fresh, f(tmp));
```

Notice that the expansion-internal binding of tmp has been alpha-varied to tmp_fresh. The reference to tmp in the spliced expression then refers, as intended, to the application site binding.

Although this strict capture avoidance discipline implies that TLMs cannot intentionally introduce bindings directly into spliced expressions, this does not imply that values cannot flow from the expansion into a spliced expression. It simply means that when this is intended, the segment type must be a function type, which serves to make this interface explicit. Reason's concise lambda notation, (x) => e, decreases the syntactic cost of this approach. For example, we cannot define list comprehension notation like the following because the binding site of x is not clear:

```
$listcomp `(x + 1 | x in lo .. hi)` /* NO! cannot reason abstractly */
```

However, the following is permitted, because x is bound by the spliced lambda expression and the corresponding segment type makes the type of the interface explicit:

```
$listcomp `((x) => x + 1 | lo .. hi)`  /* OK! */
```

Our contention is that small syntactic costs like these are more than justified by the peace of mind of knowing that unfamiliar literal notation cannot possibly be performing "magic" with the type and binding structure. We say more about the future prospect of a mechanism designed specifically for shorthand binding forms, e.g. Haskell-style **do** notation [Jones 2003], in Sec. 8.

```
{
open RegexParser;
let readsplice =
  Relit.Segment.read_to(")");
let unescape = (s) =>
  String.sub(s, 1, 1);
}
let special =
  ['\\' '.' '|' '*' '+'
   '?' '(' ')' '$']
let not_special = _#special
let escape = '\\' special
rule read =
  parse
  | "." { DOT }
  | "|" { BAR }
  | "*" { STAR }
  | "+" { PLUS }
  | "?" { QMARK }
  | "(" { LPAREN }
  | ")" { RPAREN }
  | not_special+ as s
    { STR(s) }
  | escape as s
    { STR(unescape(s)) }
  | "$("
    { SPLICED_REGEX
      (readsplice(lexbuf)) }
  | "$$("
    { SPLICED_STRING
      (readsplice(lexbuf)) }
  | eof { EOF }
```

(a) RegexLexer.mll

```
%{
open notation Relit.$proto_expr;
%}
%token DOT BAR STAR PLUS QMARK LPAREN RPAREN EOF
%token <string> STR
%token <Relit.Segment.t> SPLICED_REGEX
%token <Relit.Segment.t> SPLICED_STRING
%left BAR
%start <Relit.ProtoExpr.t> start
%%
start:
  | e = regex; EOF { e }
  | EOF { `( Regex.Empty )` }
regex:
  | DOT { `( Regex.AnyChar )` }
  | s = STR { `( Regex.Str($s `(s)`) )` }
  | r1 = regex; r2 = regex
    { `( Regex.Seq(`(r1)`, `(r2)`) )` }
  | r1 = regex; BAR; r2 = regex
    { `( Regex.Or(`(r1)`, `(r2)`) )` }
  | r = regex; STAR { `( Regex.Star(`(r)`) )` }
  | r = regex; PLUS
    { `( let r = `(r)`;
          Regex.Seq(r, Regex.Star(r)) )` }
  | r = regex; QMARK
    { `( Regex.Or(Regex.Empty, `(r)`) )` }
  | LPAREN; r = regex; RPAREN { r }
  | seg = SPLICED_REGEX
    { `( $spliced `(seg : Regex.t)` )` }
  | seg = SPLICED_STRING
    { `( Regex.Str
          ($spliced `(seg : string)`) )` }
```

(b) RegexParser.mly

Fig. 6. The lexer and parser for the `$regex` TLM from Fig. 3b. See Fig. 7 for relevant definitions from `Relit`.

2.2 Provider Perspective

Let us turn now to the perspective of the TLM provider, whose principal task is to define the lexer and parser named in the TLM definition—`RegexLexer` and `RegexParser` in Fig. 3b. Our implementation of TLMs for Reason, called `Relit`, assumes that the provider is using `ocamllex` [Leroy et al. 2014] and Menhir [Pottier and Régis-Gianas 2016] to generate the lexer and parser, respectively. Menhir is a modern derivative of Yacc [Johnson 1975; Tarditi and Appel 1990] with support for LR(1) grammars [Jourdan et al. 2012]. The Reason implementation also uses these tools. `Relit` requires that the TLM provider package the lexer and parser into an `ocamlfind` package named in the TLM definition—in the case of Fig. 3b, `regex_parser`—so that `Relit` can load them, together with their dependencies, at expansion-time. The TLM must also name the starting non-terminal—here, `start` on Line 4 of Fig. 3b. The `ocamllex` and Menhir definitions that implement `RegexLexer` and `RegexParser` appear in Fig. 6. After detailing this example in Sec. 2.2.1-2.2.2, we summarize the relevant parser correctness criteria in Sec. 2.2.3, describe how providers might use MetaOCaml to reason more precisely about type correctness in Sec. 2.2.4, and finally outline an alternative implementation strategy in Sec. 2.2.5 that sidesteps `ocamllex` and Menhir to support the use of arbitrary parse functions. In particular, we implement the `$html` TLM from Fig. 5 using an off-the-shelf HTML parser.

```
1 module Segment : {
2   type t = {start_pos: int, end_pos: int};
3   let read_to : string => Lexing.lexbuf => Segment.t; };
4 exception ExpansionError({msg: string, loc: option(Segment.t)});
5 module ProtoExpr : {
6   type t = Parsetree.expression; /* from the OCaml compiler library */
7   let spliced : Segment.t => Parsetree.core_type => t; };
8 notation $re_expr at Parsetree.expression { /* ... (see text) ... */ };
9 notation $proto_expr at ProtoExpr.t = $re_expr;
```

Fig. 7. A fragment of the `Relit` module signature relevant to expression TLM providers.

2.2.1 `RegexLexer` In most respects, the definition of the lexer in Fig. 6a is conventional. It reads various lexical patterns on Lines 8-32, emitting the corresponding tokens specified by the **`%token`** declarations in Fig. 6b, which have been brought into scope by Line 2 of Fig. 6a. Lines 22-23 combine sequences of non-special characters (e.g. alphanumeric characters) into a single string token. Lines 24-25 implement backslash-prefixed escape sequences for characters that have special meaning in this notation for regexes, emitting a string token in each instance.[1]

Lines 24-31 of the lexer, which recognize the notation for regex splicing `$(`e`)`, and string splicing, `$$(`e`)`, are more unusual. Recall from Sec. 2.1.3 that the parser must represent spliced terms abstractly, by their location within the literal body. This implies that the TLM does not need to parse the spliced expression itself, as long as it can independently decide where the spliced expression starts and ends. A paired start and end position is a segment, represented by a value of the type `Segment.t` defined in the `Relit` helper library as specified on Line 2 of Fig. 7. Lines 26-28 and Lines 29-31 of the lexer produce a segment by calling a helper function, `Relit.Segment.read_to`, that internally invokes the Reason lexer on the provided lexing buffer until it sees an instance of the provided token outside of a Reason comment, string literal or generalized literal. If the provided token is a right delimiter also used by Reason, like `")"` on Line 4 of Fig. 6a, then `read_to` looks for the first unmatched instance. For example, if the remaining lexing buffer contains `f("))"))A|G` then `read_to` will consume up until just before the final instance of `)` and emit the appropriate segment. Similar helper functions, not shown, are available for reading out single identifiers, simple paths like `X1.X2.xy` and a few other unambiguously delimited fragments of the Reason syntax.

This approach has two major benefits. First, it supports splicing even within literal notations that have a very different lexical structure from Reason itself, as in this example. Second, it allows the parser to avoiding needing to link to Reason's expression grammar (though this is supported by Menhir), which substantially simplifies reasoning about ambiguities. Spliced expressions arrive into the parser as single, opaque tokens as specified on Lines 6-7 of Fig. 6b. The system will recursively expand spliced expressions after the parser has finished generating the proto-expansion.

2.2.2 `RegexParser` and Quasiquotation The job of the parser is to generate a proto-expansion given the tokens generated by the lexer, or if this is not possible, to indicate an expansion error by raising either a Menhir parse error or `Relit.ExpansionError` (Line 4 of Fig. 7) with an appropriate error message and, if possible, an error location. In the case of expression TLMs, the proto-expansion must be a proto-expression (we consider proto-patterns in the next section). Proto-expressions are encoded as values of type `Relit.ProtoExpr.t`. This type is defined on Line 6 of Fig. 7 as a synonym for `Parsetree.expression`, which is the standard representation of expression parse trees exposed by the OCaml compiler library [Leroy et al. 2014].

[1]A better approach would be to define a second lexing rule that greedily combines sequences consisting of non-special characters and escape characters into a single STR token, but for the sake of exposition, we stick to this simpler approach.

In order to repurpose this existing parse tree representation for proto-expansions, we must provide some way to unambiguously represent splice references, notated earlier in the paper as **spliced**<m..n : ty> where m..n specifies a segment and ty is the segment's expected type. The ProtoExpr module solves this problem by providing a function spliced that takes a segment, of type Segment.t, and a type representation, of type Parsetree.core_type, which also comes from OCaml's compiler library, and produces a corresponding value of type ProtoExpr.t that uniquely represents the corresponding splice reference (see Sec. 6 for more on the internal representation).

say something about splice references in sec 6

Note that it is only because we explicitly name dependencies in TLM definitions that we can get away with using a simple, unprivileged representation of parse trees. In particular, variable renamings can be performed without needing to reason about or modify parse tree encodings in parser definitions (where variables are represented using strings, and are thus not part of the binding structure). For example, if we decided to rename the module Regex to Regexp, this renaming would need only to be performed on the module paths in the definition of $rx, as long as the name of the internal module variable in the dependencies list remains Regex.

The datatypes in OCaml's Parsetree module are necessarily intricate, given the sophistication of the OCaml system, so constructing expression encodings manually is generally too unwieldy. Addressing this class of problem is, of course, the motivation for this very paper, so it is only natural that we define TLMs for working with OCaml parse trees using familiar surface syntax, extended with splice forms. In the literature, literal notation for the host language's own parse trees is called *quasiquotation*, and splicing is referred to as *unquotation* or *antiquotation* [Bawden 1999; Mainland 2007; Shabalin et al. 2013]. The TLM $re_expr specified on Line 10 of Fig. 7 provides quasiquotation for OCaml parse trees using Reason's surface syntax. The accompanying technical report describes its implementation. For clarity, the $proto_expr TLM is defined as a synonym for $re_expr on Line 12 of Fig. 7 (the two TLMs may diverge in the future). We apply this TLM many times in the semantic actions in Fig. 6b. Notice, however, that the generalized literal forms in Fig. 6b are not each prefixed by $proto_expr. Instead, we use the **open notation** directive on Line 2, which implicitly applies $proto_expr to all unadorned generalized literal forms in its scope.

To support antiquotation, the $re_expr TLM repurposes TLM application and generalized literal forms, as can be observed throughout Fig. 6b. In particular, parse tree splicing is supported by unadorned generalized literal forms, e.g. on Line 23 where we implement the regex notation r+ in terms of sequencing and star, taking care to bind r to an expansion-internal variable—also r but distinguished by the segmentation—to avoid double evaluation. String splicing (which converts the spliced string expression into a parse tree of string constant form) is performed by repurposing the TLM name $s (Line 16). There is also notation for references to spliced expressions that repurposes the TLM name $spliced, as seen on Lines 28-32, where the regex splicing logic is implemented.

TLM-related forms are available to be repurposed in this way by $proto_expr because there is no reasonable way for $re_expr to support antiquotation across notational boundaries. For example, if $re_expr had instead used some other antiquotation notation, e.g. ^(*e*), then consider this example:

```
$proto_expr `( let y = ^(r1); $m `(...{^(r2)}...)` )`
```

The sequence ^(r1) is in expression position, so antiquotation occurs. However, there is no way for $re_expr to know how the TLM that $m will eventually refer to will parse the sequence ^(r2)—it may end up in a spliced expression, or it may have some other meaning—so it cannot perform antiquotation. In other words, TLM expansion is fundamentally "outside-in". Notice that the segmentation does communicate the fact that r2 has not been antiquoted (it remains lavender). However, to avoid confusion, we decided not to support the generation of TLM application forms via $re_expr, and instead repurposed that notation for antiquotation.

This issue does **not** come up when using TLMs to define quotation literals for languages other than Reason itself (an example is given in Sec. 3), so if this design proves too limiting, it may be reasonable to build "inside-out" quasiquotation primitively into Reason. In fact, this is already possible using a preprocessor, `ppx_metaquot`, from the `ppx_tools` library [Frisch et al. 2017], or with MetaOCaml and related systems, discussed below.

2.2.3 Correctness Criteria In order to maintain the abstract reasoning principles discussed in Sec. 2.1, the system *validates* each proto-expansion generated by the parser as follows:

(1) First, the segmentation is computed and validated. This involves checking two criteria:
  (a) The segments must be in-bounds, of positive extent, non-overlapping and separated by at least one character.
  (b) The segment types must encode valid types assuming only the TLM dependencies.
(2) Second, the proto-expansion is typechecked assuming only the TLM dependencies (plus opened `Pervasives`), treating each splice reference as a variable of the specified segment type.

If proto-expansion validation fails, the client is notified that the applied TLM is incorrectly implemented. The proto-expansion validation criteria are exactly the correctness criteria that parser writers must consider, quantifying over all possible literal bodies for which a proto-expansion is generated, in order to avoid exposing this class of compile-time error to clients.

2.2.4 MetaOCaml In "vanilla" OCaml, it is not possible to express a datatype that classifies only valid proto-expansions according to the correctness criteria above. The MetaOCaml type system comes closer by extending the semantics of OCaml with values of type `'a code`, which are constructed using a primitive typed quasiquotation operation [Kiselyov 2014; Taha 2004]. In MetaOCaml-BER, which is an active implementation of (an extension of) MetaOCaml as a fork of the compiler, and `ppx_stage`, which implements a subset of MetaOCaml as a compiler plugin [Dolan 2018], a value of type `'a code` is represented as, and can be coerced safely to, an OCaml parse tree for the corresponding expression of type `'a` [Kiselyov 2014]. It is therefore possible to implement a TLM parser using `'a code` values internally, doing the coercion only at the end. However, the typing guarantee is relative to the context where the quotation was constructed, i.e. the parser's implementation, so a validation step is still needed to ensure that the assumptions are consistent with those specified by the **`dependencies`** clause.

The segmentation criteria cannot be statically enforced and, in both mentioned implementations of MetaOCaml, it is impossible to express splice references directly because they do not yet support terms that contain explicit type annotations. There are various workarounds, e.g. by defining for each possible segment type `t` a dummy value of type `t code` and then post-processing the parse tree after the coercion step. Overall, however, this approach can substantially increase a TLM provider's confidence in the type correctness of the expansion logic, with the caveats just mentioned. With `ppx_stage`, it is possible to compile the parser separately without requiring that clients use the plugin. TLMs can be applied inside quoted code as long as the applied TLM is accessed through a dependency of the TLM being defined (so that internal dependency references correctly resolve).

2.2.5 Other Implementation Strategies Although `ocamllex` and Menhir are powerful tools, they are not right for every parsing job. For example, we might want to use a different parser generator, a parser combinator library [Hutton 1992], or post-process the result of calling an existing parser to produce a corresponding proto-expansion. Fortunately, it is easy to bypass `ocamllex` and Menhir: the `Relit` library includes a trivial lexer, `Relit.TrivLexer`, and a few functors that take care of the necessary boilerplate in situations where we simply want to define the parser as a function of type **`string`** `=> Relit.ProtoExpr.t`. For example, the accompanying technical report briefly describes how `$re_expr` can be expressed in terms of the Reason parser, a parse tree serializer

```
1 module Lambda = {/* a typical encoding */};
2 module LambdaNotation {
3   notation $term at Lambda.term {
4     lexer SimpleLangLexer;
5     expression parser LambdaParser.term_e;
6     pattern parser LambdaParser.term_p;
7     dependencies
8       { module Lambda = Lambda } };
9   notation $v at Lambda.v {/* analogous */}
10 };
```

(a) Expression and pattern TLMs for Lambda terms and values

```
1 open LambdaNotation;
2 exception Unbound(Lambda.var);
3 let rec eval = $term.(fun
4 | `(x)` => raise Unbound(x)
5 | `(lam x.e)` => $v `(lam x.e)`
6 | `(e1(e2))` => {
7   let $v `(lam x.e)` = eval(e1);
8   let v2 = eval(e2);
9   eval(`([v2/x]e)`)
10 });
```

(b) A reasonably elegant Lambda evaluator

Fig. 8. Expression and pattern literal notation for lambda terms and values

and a tree transformation. It also details how the `$html` TLM shown previously is implemented using an existing, "production grade" HTML parsing library, `Markup.ml` [Bachin 2018], together with a simple post-processing step.

3 PATTERN LITERALS

The previous section introduced expression TLMs, which support value construction. This section introduces pattern TLMs, which support value deconstruction via the structural pattern matching facilities common to ML-like languages.

For example, the definitions outlined in Fig. 8a allow us to elegantly express an evaluator for the lambda calculus as shown in Fig. 8b. We assume that the terms of `Lambda` are represented differently from the values to avoid needing additional "impossible" cases, but we can use the same notation for both without conflict by defining separate TLMs, `$term` and `$v`, respectively. Each of these TLMs defines both expression and pattern literal notation, distinguished by qualifiers in the TLM definition as shown on Lines 5-6 of Fig. 8a, and we see examples of all four of these notations in use in Fig. 8b. These TLMs were designed to be used for operating on `Lambda` terms, so we chose a splicing convention inspired by the typical convention "on paper", where identifiers appearing in the literal are parsed as spliced (OCaml) variables, as indicated by the colors in Fig. 8b. For example, on Line 5, `x` and `e` are bound by the literal pattern for `Lambda` terms on the left, then spliced into the expression literal on the right to construct the corresponding `Lambda` value.

From a client programmer's perspective, reasoning principles analogous to those described in Sec. 2.1 for expression TLMs are available. **Responsibility** is assigned by the same protocol, and the type annotation on the responsible TLM governs the type of the generated pattern, satisfying the **Expansion Typing** condition. Patterns can contain module paths and type annotations, so pattern TLMs are governed by the same **Context Independence** condition as expression TLMs. The same protocol around **Segmentation** is also enforced.

The main novelty has to do with **Segment Typing** and **Capture**. Variables in patterns do not refer to existing bindings, as in expressions, but rather introduce bindings into other expressions, e.g. the corresponding branch of a case analysis or the body of a function, so the critical question is this: given only the segmentation of a pattern literal, can we determine exactly which variables the expansion binds, and what the types of these variables are? To answer in the affirmative, we need to preclude the possibility of "invisible bindings", so the system ensures that pattern literals bind only those variables that appear inside spliced patterns. This implies that *proto-patterns simply cannot contain pattern variables*. The segmentation assigns a type to each spliced segment, so the question just asked can be answered given only the segmentation.

```
module Map = {
  module type S = {
    type key;
    type t('a);
    let empty : t('a);
    let add :
      key => 'a => t('a) => t('a);
  /* ... */ }
}
```

(a) The `Map.S` signature

```
notation $map(M : Map.S, type a) at M.t(a) {
  lexer  MapLexer; parser MapParser;
}
```

(b) The parametric TLM `$map`

```
1 notation $stringmap = $map(Misc.StringMap);
2 let make_request id req v = $stringmap
3   `( "session_id" -> string_of_int(id),
4               req -> v )`;
```

(c) Applying `$map`

Fig. 9. Literal notation for all finite map implementations using parametric TLMs.

From the provider's perspective, the parser specified by the **`pattern parser`** clause must generate a proto-pattern, rather than a proto-expression. In Reason, this is a value of type `Relit.ProtoPat.t`. Much as with `Relit.ProtoExpr.t` from Fig. 7, this type is defined as a synonym for `Parsetree.pattern` from OCaml's compiler library equipped with an additional function, `Relit.ProtoPat.spliced`, that constructs splice references. In principle, it might be better to define `Relit.ProtoPat.t` to exclude the representation of pattern variables, and we do just that in the theoretical development in Sec. 5, but `Relit` checks this straightforward condition during proto-expansion validation. Note that in OCaml, boolean guards can be associated with rules of a **`match`** (in Reason, **`switch`**) expression, but the guard is not part of the syntax of patterns. (If it were, we would need to validate it as in Sec. 2).

4 PARAMETRIC TLMS

All of the examples that we have discussed so far operate at a single specified type. This section introduces *parametric TLMs*, which can take type and module parameters and operate over a type- and module-parameterized family of types.

Fig. 9a shows a portion of the signature `Map.S` from the OCaml standard library, which specifies a polymorphic abstract data type [Harper 1997; Liskov and Zilles 1974] `t('a)` of finite maps with keys of type `key`. There are many ways to implement this signature. For example, the `Map.Make` functor in the standard library implements this signature using balanced binary trees given a module that specifies a `key` type equipped with comparison functions. The `Misc.StringMap` module in the standard library also provides a specialized implementation of this signature with type `key =` **`string`**.

We can define a TLM, `$map`, that is parametric over implementations of this signature, `M : Map.S`, and over choices of the co-domain type, `a`, as shown in Fig. 9b. We can then partially apply `$map` to `Misc.StringMap` as shown on Line 1 of Fig. 9c, producing a TLM, `$stringmap`, parameterized only over the type `a`, with expansion type `Misc.StringMap.t(a)`. We can apply `$stringmap` to a type specialize it further, e.g. `$stringmap` **`string`** has expansion type `Misc.StringMap.t(`**`string`**`)`. For TLMs where all remaining parameters are types mentioned in the expansion type, we can also immediately apply the TLM to a generalized literal form as shown on Lines 2-3 of Fig. 9c. The type parameters are determined by unification using the types inferred for the spliced expressions together with the segment types specified in the segmentation, and any surrounding type constraints. In this example, we can infer `a` to be **`string`** because `string_of_int(id) :` **`string`** and the corresponding segment type is `a`. TLM abbreviations can themselves be parameterized to support partial application of parameters other than the last.

The proto-expansion generated by a parametric TLM can refer to its parameters. Validation checks that the proto-expansion is truly parametric—in this case, the proto-expansion must be valid for all modules `M : Map.S` and types `a`. It is only after validation that we substitute the actual parameters, here `Misc.StringMap` for `M` and **`string`** for `a`, into the final expansion.

The **`dependencies`** clause shown in previous examples can be understood in terms of parameterization over the dependencies, using the given module variables, followed immediately by partial application of the corresponding module paths. Note, however, that module parameters need an explicit signature, like functor (module function) arguments in ML.

An alternative point-of-view in ML is to treat the **`dependencies`** clause as fundamental and achieve parameterization using module functions (functors), e.g. we could express `$map` as follows:

```
module MapNotation = (M : Map.S, A : { type a; }) => {
  notation $map at M.t(A.a) {
    lexer MapLexer; parser MapParser;
    dependencies { module M = M; type a = A.a; } } }
```

5 TYPED LITERAL MACROS, FORMALLY

This section will present a calculus of simple expression and pattern TLMs based on "core ML" called $\mathbf{ML}^{\text{Lit}}$. By the end of this section, we will have a theorem that formalizes the six reasoning principles that were outlined informally in the previous sections. $\mathbf{ML}^{\text{Lit}}$ consists of an *unexpanded language*, or *UL*, defined by typed expansion to the core language, named for the purposes of this paper **ML**.

5.1 Core Language

Fig. 10 gives the syntax of the core language, **ML**. This language forms a standard pure functional language with partial function types, quantification over types, recursive types, labeled product types, labeled sum types and support for pattern matching.[2] Formally, core language terms are abstract binding trees (ABTs) identified up to alpha-equivalence, so we follow the syntactic conventions of Harper [2012]. This notational convention also clearly distinguishes core terms from unexpanded terms, which formally behave quite differently as we will describe below.

The reader is directed to *PFPL* [Harper 2012] for a detailed introductory account of all of these language constructs. We will only tersely summarize the static and dynamic semantics of the core language because the particularities are not critical to our ideas. The static semantics is organized around the type formation judgement, $\Delta \vdash \tau$ type, the expression typing judgement, $\Delta\, \Gamma \vdash e : \tau$, and the pattern typing judgement, $p : \tau \dashv\!\!\!\dashv \Gamma$. Type formation contexts, Δ, track hypothesis of the form t type, and typing contexts, Γ, track hypotheses of the form $x : \tau$. In the pattern typing judgement, Γ collects the typing hypotheses generated by p. These judgements are inductively defined in the accompanying technical report along with necessary auxiliary structures and standard lemmas. The dynamic semantics of **ML** is organized around the judgements e val, which says that e is a value, and $e \Downarrow e'$, which says that e evaluates to the value e'. As in ML, evaluation can diverge (general recursion is possible via recursive types [Harper 2012]). It can also result in match failure.

5.2 Syntax of the Unexpanded Language

Fig. 11 defines the syntax of the unexpanded language. Unlike core language types and expressions, unexpanded expressiona and types are **not** abstract binding trees – we do **not** assume the standard notions of renaming, alpha-equivalence or substitution. Instead, they are simple inductive structures. This is because unexpanded expressions remain "partially parsed" due to the presence of literal

[2]The accompanying technical report shows how to remove pattern matching (and pattern TLMs) from the calculus by adding the usual elimination forms, e.g. `unfold`(e) rather than the `foldp`(p) pattern for values of recursive type.

$$
\begin{array}{llll}
\text{Typ} & \tau & ::= & t \mid \mathtt{parr}(\tau;\tau) \mid \mathtt{all}(t.\tau) \mid \mathtt{rec}(t.\tau) \mid \mathtt{prod}(\{i \hookrightarrow \tau_i\}_{i\in L}) \mid \mathtt{sum}(\{i \hookrightarrow \tau_i\}_{i\in L}) \\
\text{Exp} & e & ::= & x \mid \mathtt{lam}\{\tau\}(x.e) \mid \mathtt{ap}(e;e) \mid \mathtt{tlam}(t.e) \mid \mathtt{tap}\{\tau\}(e) \mid \mathtt{fold}(e) \mid \mathtt{tpl}(\{i \hookrightarrow e_i\}_{i\in L}) \mid \mathtt{inj}[\ell](e) \\
 & & \mid & \mathtt{match}(e;\{r_i\}_{1\le i\le n}) \\
\text{Rule} & r & ::= & \mathtt{rule}(p.e) \\
\text{Pat} & p & ::= & x \mid \mathtt{wildp} \mid \mathtt{foldp}(p) \mid \mathtt{tplp}(\{i \hookrightarrow p_i\}_{i\in L}) \mid \mathtt{injp}[\ell](p)
\end{array}
$$

Fig. 10. Syntax of the core language, **ML**, which is an entirely standard typed lambda calculus. Metavariable x ranges over variables, t over type variables, ℓ over labels and L over finite sets of labels. We write $\{i \hookrightarrow \tau_i\}_{i\in L}$ for a finite mapping of each label i in L to some type τ_i, and similarly for other sorts of terms. We write $\{r_i\}_{1\le i\le n}$ for a finite sequence of $n > 0$ rules.

$$
\begin{array}{llll}
\text{UTyp} & \hat\tau & ::= & \hat t \mid \hat\tau \rightharpoonup \hat\tau \mid \forall \hat t.\hat\tau \mid \mu \hat t.\hat\tau \mid \langle\{i \hookrightarrow \hat\tau_i\}_{i\in L}\rangle \mid [\{i \hookrightarrow \hat\tau_i\}_{i\in L}] \\
\text{UExp} & \hat e & ::= & \hat x \mid \lambda \hat x{:}\hat\tau.\hat e \mid \hat e(\hat e) \mid \Lambda \hat t.\hat e \mid \hat e[\hat\tau] \mid \mathtt{fold}(\hat e) \mid \langle\{i \hookrightarrow \hat e_i\}_{i\in L}\rangle \mid \mathtt{inj}[\ell](\hat e) \\
 & & \mid & \mathtt{match}\ \hat e\ \{\hat r_i\}_{1\le i\le n} \\
 & & \mid & \mathtt{notation}\ \hat a\ \mathtt{at}\ \hat\tau\ \{\ \mathtt{expr\ parser}\ e; \mathtt{expansions\ require}\ \hat e\ \}\ \mathtt{in}\ \hat e \\
 & & \mid & \mathtt{notation}\ \hat a\ \mathtt{at}\ \hat\tau\ \{\ \mathtt{pat\ parser}\ e\ \}\ \mathtt{in}\ \hat e \\
 & & \mid & \hat a\ \text{‘}(b)\text{‘} \\
\text{URule} & \hat r & ::= & \hat p \Rightarrow \hat e \\
\text{UPat} & \hat p & ::= & \hat x \mid \_ \mid \mathtt{fold}(\hat p) \mid \langle\{i \hookrightarrow \hat p_i\}_{i\in L}\rangle \mid \mathtt{inj}[\ell](\hat p) \mid \hat a\ \text{‘}(b)\text{‘}
\end{array}
$$

Fig. 11. Syntax of the $\mathbf{ML}^{\mathsf{Lit}}$ unexpanded language (UL). Metavariable $\hat t$ ranges over type identifiers, $\hat x$ over expression identifiers, $\hat a$ over TLM identifiers and b over literal bodies.

$$
\dfrac{\dfrac{}{\hat\Delta \vdash \hat\tau \rightsquigarrow \tau\ \mathsf{type}} \quad \dfrac{\dfrac{}{\hat\Delta \vdash \hat\tau \rightsquigarrow \tau\ \mathsf{type}} \quad \dfrac{}{\hat\Delta\ \langle \hat x \rightsquigarrow x_2; x_1 : \tau, x_2 : \tau\rangle \vdash_{\hat\Psi;\hat\Phi} \hat x \rightsquigarrow x_2 : \tau}\ \textsc{ee-id}}{\hat\Delta\ \langle \hat x \rightsquigarrow x_1; x_1 : \tau\rangle \vdash_{\hat\Psi;\hat\Phi} \lambda \hat x{:}\hat\tau.\hat x \rightsquigarrow \mathtt{lam}\{\tau\}(x_2.x_2) : \mathtt{parr}(\tau;\tau)}\ \textsc{ee-lam}}{\hat\Delta\ \langle\emptyset;\emptyset\rangle \vdash_{\hat\Psi;\hat\Phi} \lambda \hat x{:}\hat\tau.\lambda \hat x{:}\hat\tau.\hat x \rightsquigarrow \mathtt{lam}\{\tau\}(x_1.\mathtt{lam}\{\tau\}(x_2.x_2)) : \mathtt{parr}(\tau;\mathtt{parr}(\tau;\tau))}\ \textsc{ee-lam}
$$

Fig. 12. An example expansion derivation demonstrating how identifiers and variables are separately tracked.

bodies, b, from which spliced terms might be extracted during expansion. In fact, unexpanded types and expressions do not involve variables at all, but rather *identifiers*, $\hat t$ and $\hat x$.

There is also a corresponding context-free textual syntax for the UL. Giving a complete definition of the context-free textual syntax with, for example, a context-free grammar is not critical to our purpose. Instead, we only posit partial metafunctions parseUTyp(b), parseUExp(b) and parseUPat(b) that go from character sequences, b, to unexpanded types, expressions and patterns, respectively.

5.3 Typed Expansion

The *typed expansion judgements* below specify the expansion process, which in the setting of $\mathbf{ML}^{\mathsf{Lit}}$ occurs simultaneously with typing (see Sec. 6 for certain nuances in our implementation).

$$
\begin{array}{ll}
\hat\Delta \vdash \hat\tau \rightsquigarrow \tau\ \mathsf{type} & \hat\tau \text{ has well-formed expansion } \tau \\
\hat\Delta\ \hat\Gamma \vdash_{\hat\Psi;\hat\Phi} \hat e \rightsquigarrow e : \tau & \hat e \text{ has expansion } e \text{ of type } \tau \\
\hat\Delta \vdash_{\hat\Phi} \hat p \rightsquigarrow p : \tau \dashv\!\!\mid \hat\Gamma & \hat p \text{ has expansion } p \text{ matching } \tau
\end{array}
$$

Most of the unexpanded forms in Figure 11 mirror the expanded forms. We refer to these as the *common forms*. The typed expansion rules that handle common forms mirror the corresponding typing rules. The *expression TLM context*, $\hat\Psi$, and the *pattern TLM context*, $\hat\Phi$, detailed below pass through these rules opaquely. For example, the rules for variables and lambdas are shown being applied in the example derivation in Fig. 12, discussed below. The full set of rules is in the accompanying technical report.

The only technical subtlety related to common forms has to do with the relationship between identifiers, $\hat x$, in the UL and variables, x, in the core language. We might hope to identify identifiers

EE-DEF-SETLM

$$\frac{\begin{array}{c}\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \ \mathsf{type} \\ \emptyset\ \emptyset \vdash e_{\mathrm{parse}} : \mathtt{parr(Body;ParseResultE)} \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_{\mathrm{dep}} \rightsquigarrow e_{\mathrm{dep}} : \tau_{\mathrm{dep}} \\ \hat{\Gamma} = \langle \mathcal{G}; \Gamma \rangle \qquad \hat{\Delta}\ \langle \mathcal{G}; \Gamma, x : \tau_{\mathrm{dep}} \rangle \vdash_{\hat{\Psi}, \hat{a} \rightsquigarrow x \hookrightarrow \mathtt{setlm}(\tau; e_{\mathrm{parse}}); \hat{\Phi}} \hat{e} \rightsquigarrow e : \tau' \\ e_{\mathrm{defn}} = \mathtt{ap(lam\{}\tau_{\mathrm{dep}}\mathtt{\}(}x.e\mathtt{)};e_{\mathrm{dep}}\mathtt{)}\end{array}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{notation}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \{\ \mathtt{expr\ parser}\ e_{\mathrm{parse}}; \mathtt{expansions\ require}\ \hat{e}_{\mathrm{dep}}\ \}\ \mathtt{in}\ \hat{e} \rightsquigarrow e_{\mathrm{defn}} : \tau'}$$

EE-DEF-SPTLM

$$\frac{\begin{array}{c}\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \ \mathsf{type} \qquad \emptyset\ \emptyset \vdash e_{\mathrm{parse}} : \mathtt{parr(Body;ParseResultP)} \\ \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}, \hat{a} \rightsquigarrow \_ \hookrightarrow \mathtt{sptlm}(\tau; e_{\mathrm{parse}})} \hat{e} \rightsquigarrow e : \tau'\end{array}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{notation}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \{\ \mathtt{pat\ parser}\ e_{\mathrm{parse}}\ \}\ \mathtt{in}\ \hat{e} \rightsquigarrow e : \tau'}$$

Fig. 13. The typed expansion rules for expression and pattern TLM definitions.

in the UL with variables in the core language and track the bindings in unexpanded terms using typing contexts as defined in the core language, but we cannot because the only operation for producing a new typing context from an existing typing context is context extension, written $\Gamma, x : \tau$, which is defined only when $x \notin \mathrm{dom}(\Gamma)$. When working with abstract binding trees, i.e. terms identified up to variable renaming, we typically need to give no thought to this condition because it is always possible to implicitly rename the term under consideration when shadowing occurs to discharge this requirement. However, we cannot implicitly rename unexpanded terms. Changing the definition of typing contexts would have significant implications throughout the metatheory of the core language, which we seek to avoid touching.

Instead, we define *unexpanded typing contexts*, $\hat{\Gamma}$, as pairs of the form $\langle \mathcal{G}; \Gamma \rangle$, where $\mathcal{G}$ maps each expression identifier $\hat{x} \in \mathrm{dom}(\mathcal{G})$ to a variable, x, written $\hat{x} \rightsquigarrow x$. The typing context, Γ, only tracks the type of these variables. We define the identifier update operation $\mathcal{G} \uplus \hat{x} \rightsquigarrow x$ as producing the expression identifier expansion context that maps $\hat{x}$ to x, written $\hat{x} \rightsquigarrow x$, and defers to $\mathcal{G}$ for all other expression identifiers, with no requirement that $\hat{x}$ be apart from $\mathrm{dom}(\mathcal{G})$. We define $\hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau$ when $\hat{\Gamma} = \langle \mathcal{G}; \Gamma \rangle$ as an abbreviation of $\langle \mathcal{G} \uplus \hat{x} \rightsquigarrow x; \Gamma, x : \tau \rangle$. Unexpanded type formation contexts, $\hat{\Delta}$, are analogous.

To develop an intuition for how this formulation solves the problem, it is instructive to inspect the derivation in Fig. 12 of the expansion of the unexpanded expression $\lambda\hat{x}{:}\hat{\tau}.\lambda\hat{x}{:}\hat{\tau}.\hat{x}$ assuming $\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau$ type. Notice that each time Rule EE-LAM is applied, the type identifier expansion context is updated but the typing context is extended with a fresh variable, first x_1 then x_2, which is possible by alpha varying only the expansion, leaving the unexpanded term unchanged.

5.3.1 TLM Definitions There are four unexpanded forms that are not common forms – the two TLM definition forms and the two TLM application forms. Let us start with TLM definitions. Rule EE-DEF-SETLM in Fig. 13 governs simple expression TLM (seTLM) definitions.

The first premise expands the unexpanded expansion type, $\hat{\tau}$, producing the expansion type, τ.

The second premise checks that the parse function, e_{parse}, is a closed expanded function with input type Body and return type `ParseResultE`.[3] The type abbreviated Body classifies encodings of literal bodies, b. Rather than defining Body explicitly it suffices to take as a condition that there is an isomorphism between literal bodies and values of type Body mediated in one direction by a judgement $b \downarrow_{\mathsf{Body}} e_{\mathrm{body}}$ that is used in the rule for TLM application discussed below. The return type, `ParseResultE`, abbreviates a labeled sum type, `sum(Error` $\hookrightarrow \langle\rangle$, `SuccessE` $\hookrightarrow$ `PrExpr)`, that allows the TLM's parser to distinguish parse errors from successful parses.[4]

[3]Think of e_{parse} as the result of parser name resolution, which produces a "compiled"—i.e. closed and expanded—term.
[4]In `Relit`, we used an exception, `Relit.ExpansionError`, rather than a sum type for the same purpose.

The type abbreviated `PrExpr` classifies encodings of *proto-expressions*, $\grave{e}$ (pronounced "grave e"). The syntax of proto-expressions, defined in Fig. 15, will be described when we describe proto-expansion validation in Sec. 5.4. As with `Body`, we need only take as a condition that there is an isomorphism between values of type `PrExpr` and closed proto-expressions, which is mediated in one direction by the *proto-expression decoding judgement*, $e \uparrow_{\texttt{PrExpr}} \grave{e}$ that we will return to when we describe TLM application below (see accompanying technical report for the full condition).

In $\mathbf{ML}^{\mathsf{Lit}}$, we model the **dependencies** clause as specifying a single value dependency. The third premise of Rule EE-DEF-SETLM generates the expanded dependency, e_{dep} of type τ_{dep}, from the unexpanded dependency, $\hat{e}_{\text{dep}}$. Note that we do not need to explicitly allow for type dependencies (which would be formally cumbersome because we cannot "tuple together" types like we can values in this setting). Instead, type dependencies can be expressed by dependency on a value of existential type to be unpacked by the expansion. This existential type can in turn be expressed in terms of universal types by the well-known encoding [Harper 2012; Reynolds 1983]. This workaround is perhaps unsurprising given that existentials relate closely to modules, which package both types and values [Harper 2012; Mitchell and Plotkin 1988].

Having processed the TLM definition, we are ready to continue into the unexpanded expression $\hat{e}$ where the TLM is bound. To activate the TLM definition for use by $\hat{e}$, the third row of premises in Rule EE-DEF-SETLM first generates a fresh variable, x, to stand for the value dependency. We will use this variable to instantiate the dependency in each generated expansion when we discuss TLM application below. Note that there is no corresponding expression identifier in $\mathcal{G}$. Second, the rule extends the expression TLM context, $\hat{\Psi}$, to associate the TLM identifier $\hat{a}$ with x, as well as with the given expansion type and parse function. If $\hat{a}$ was already defined, the previous definition is shadowed (the accompanying technical report gives some additional details on how TLM contexts are structured).

The final premise of Rule EE-DEF-SETLM, together with the conclusion of the rule, specifies the expansion of the TLM definition as being of function application form—it wraps e, where the variable x stands free for the dependency, with a lambda binding x, and then immediately applies it to pass down the actual value dependency, e_{dep}. In other words, it **let**-binds the dependency. This defers to the core language with regard to whether e_{dep} is evaluated eagerly or lazily.

Rule EE-AP-SPTLM for pattern TLM definitions shown in Fig. 13 is analogous but simpler, because in $\mathbf{ML}^{\mathsf{Lit}}$ patterns are entirely structural (there are no module paths that they might depend on).

5.3.2 TLM Application The unexpanded expression form for applying an seTLM identified as $\hat{a}$ to a literal form with literal body b is $\hat{a}$ `‘(`b`)‘`. Rule EE-AP-SETLM governing this form is in Fig. 14.

The first two premises serve simply to "look up" $\hat{a}$ in the expression TLM context, $\hat{\Psi}$, and to look up the correspondency dependency variable, x, in $\hat{\Gamma}$.

The third premise encodes the literal body, e_{body}, producing a value e_{body} of type `Body` according to the body encoding judgement $b \downarrow_{\texttt{Body}} e_{\text{body}}$ described in Sec. 5.3.1 above.

The fourth premise applies the parse function e_{parse} to the encoding of the literal body. If parsing succeeds, i.e. a value of the form `inj[SuccessE](`e_{proto}`)` results from evaluation, then e_{proto} will be a value of type `PrExpr` (by type safety and the canonical forms lemma, assuming a well-formed expression TLM context). We call e_{proto} the *encoding of the proto-expansion*. If the parse function produces a value labeled `Error`, then typed expansion fails and formally, no rule is necessary.

The fifth premise decodes the encoding of the proto-expansion using the judgement described in Sec. 5.3.1, producing the proto-expansion itself, $\grave{e}$.

The final two premises validate the proto-expansion. Proto-expansion validation is described in Sec. 5.4 below. In $\mathbf{ML}^{\mathsf{Lit}}$, the proto-expansion does not expand directly to the final expansion but to a function, here e, from the dependency type, τ_{dep}, to the expansion type, τ, as suggested by

$$\frac{\begin{array}{c}\text{EE-AP-SETLM}\\ \hat{\Psi} = \hat{\Psi}', \hat{a} \rightsquigarrow x \hookrightarrow \texttt{setlm}(\tau; e_{\text{parse}}) \qquad \hat{\Gamma} = \langle \mathcal{G}; \Gamma, x : \tau_{\text{dep}} \rangle \\ b \downarrow_{\text{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj[SuccessE]}(e_{\text{proto}}) \qquad e_{\text{proto}} \uparrow_{\text{PrExpr}} \grave{e} \\ \text{seg}(\grave{e}) \text{ segments } b \qquad \emptyset\ \emptyset \vdash^{\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; \hat{\Phi}; b} \grave{e} \rightsquigarrow e : \texttt{parr}(\tau_{\text{dep}}; \tau)\end{array}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{a}\ \texttt{`}(b)\texttt{`} \rightsquigarrow \texttt{ap}(e; x) : \tau}$$

$$\frac{\begin{array}{c}\text{PE-AP-SPTLM}\\ \hat{\Phi} = \hat{\Phi}', \hat{a} \rightsquigarrow \_ \hookrightarrow \texttt{sptlm}(\tau; e_{\text{parse}}) \\ b \downarrow_{\text{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj[SuccessP]}(e_{\text{proto}}) \qquad e_{\text{proto}} \uparrow_{\text{PrPat}} \grave{p} \\ \text{seg}(\grave{p}) \text{ segments } b \qquad \grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\hat{\Delta}; \hat{\Phi}; b} \hat{\Gamma}\end{array}}{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{a}\ \texttt{`}(b)\texttt{`} \rightsquigarrow p : \tau \dashv\!\mid \hat{\Gamma}}$$

Fig. 14. The typed expansion rules for expression and pattern TLM application.

the right-hand side of the final premise. In the conclusion of the rule, we apply the dependency variable, x, to produce the final expansion.[5]

The typed pattern expansion rule governing pattern TLM application, Rule PE-AP-SPTLM in Fig. 14, is analogous but again simpler because there is no need to pass dependencies into patterns in $\mathbf{ML}^{\text{Lit}}$.[6] We describe proto-pattern validation in Sec. 5.4 below.

5.4 Proto-Expansion Validation

Proto-expansion validation occurs in two steps, corresponding to the final two premises of the TLM application rules in Fig. 14.

The first of these two premises determines the segmentation of the proto-expansion by pulling out the splice references and ensures that it is valid via the predicate ψ segments b, where ψ is the segmentation. This checks that each segment has positive length and is within bounds of b, and that the segments do not overlap and are separated by at least one character (see accompanying technical report).

The second of these two premises typechecks the proto-expression or proto-pattern, and simultaneously generates a corresponding core language expression or pattern, from which the final expansion is constructed as described above. This step is specified by the following judgements:

$$\begin{array}{ll} \Delta \vdash^{\mathbb{T}} \grave{\tau} \rightsquigarrow \tau \text{ type} & \grave{\tau} \text{ has well-formed expansion } \tau \\ \Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau & \grave{e} \text{ has expansion } e \text{ of type } \tau \\ \grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\mathbb{P}} \hat{\Gamma} & \grave{p} \text{ has expansion } p \text{ matching } \tau \end{array}$$

The purpose of the *splicing scenes* $\mathbb{T}$, $\mathbb{E}$ and $\mathbb{P}$ is to "remember" the contexts and literal body from the TLM application site (cf. the final premise of Rule EE-AP-SETLM in Fig. 13) for when validation encounters spliced terms. For example, *expression splicing scenes*, $\mathbb{E}$, are of the form $\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; \hat{\Phi}; b$.

Common Forms Most of the proto-expansion forms, including all of those elided in Fig. 15 mirror corresponding expanded forms. The rules governing proto-expansion validation for these common forms in the accompanying technical report correspondingly mirror the typing rules. Splicing scenes—$\mathbb{E}$, $\mathbb{T}$ and $\mathbb{P}$–pass opaquely through these rules, i.e. none of these rules can access the application site contexts. Notice that the initial expansion-interal typing contexts, Δ and Γ, start

[5]In `Relit`, the dependency is always on the singleton module induced by the **dependencies** clause, which is **open**ed immediately, so the necessary boilerplate is inserted automatically.

[6]If there were a need to pass dependencies into patterns, we would need either pattern functions or to use an encoding trick, e.g. by requiring the result be a function containing a "dummy" match with one pattern.

$$
\begin{array}{llll}
\mathsf{PrTyp} & \grave{\tau} & ::= & t \mid \mathtt{prparr}(\grave{\tau};\grave{\tau}) \mid \cdots \mid \mathtt{splicedt}[m;n] \\
\mathsf{PrExp} & \grave{e} & ::= & x \mid \mathtt{prlam}\{\grave{\tau}\}(x.\grave{e}) \mid \mathtt{prap}(\grave{e};\grave{e}) \mid \cdots \mid \mathtt{prmatch}(\grave{e};\{\grave{r}_i\}_{1\le i\le n}) \mid \mathtt{splicede}[m;n;\grave{\tau}] \\
\mathsf{PrRule} & \grave{r} & ::= & \mathtt{prrule}(p.\grave{e}) \\
\mathsf{PrPat} & \grave{p} & ::= & \mathtt{prwildp} \mid \cdots \mid \mathtt{splicedp}[m;n;\grave{\tau}]
\end{array}
$$

Fig. 15. Syntax of proto-expansions. Proto-expansion terms are ABTs identified up to alpha-equivalence.

out empty in the rules in Fig. 14. Together, this maintains context independence (defined formally below).

Notice that proto-rules, $\grave{r}$, involve expanded patterns, p, not proto-patterns, $\grave{p}$, because proto-rules appear in proto-expressions, which are generated by expression TLMs. Proto-patterns arise only from pattern TLMs. There is no variable proto-pattern form, for the reasons described in Sec. 3.

Splice References The only interesting forms in Fig. 15 are the references to spliced types, expressions and patterns. Let us consider the rule for references to spliced expressions:

PEV-SPLICED

$$
\frac{\begin{array}{c}\text{parseUExp}(\text{subseq}(b;m;n)) = \hat{e} \qquad \emptyset \vdash^{\langle \mathcal{D};\Delta_{\text{app}}\rangle;\, b} \grave{\tau} \leadsto \tau \text{ type} \qquad \langle \mathcal{D};\Delta_{\text{app}}\rangle\ \langle \mathcal{G};\Gamma_{\text{app}}\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e : \tau \\ \Delta \cap \Delta_{\text{app}} = \emptyset \qquad \text{dom}(\Gamma) \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset\end{array}}{\Delta\ \Gamma \vdash^{\langle \mathcal{D};\Delta_{\text{app}}\rangle;\, \langle \mathcal{G};\Gamma_{\text{app}}\rangle;\, \hat{\Psi};\, \hat{\Phi};\, b} \mathtt{splicede}[m;n;\grave{\tau}] \leadsto e : \tau}
$$

This first premise of this rule parses out the requested segment of the literal body, b, to produce an unexpanded expression, $\hat{e}$. The second premise performs proto-type expansion on the given type annotation, $\grave{\tau}$, producing a type, τ. The expansion-internal type variables in Δ are not available to τ. The third premise then invokes type expansion on $\hat{e}$ *under the application site contexts*, $\langle \mathcal{D};\Delta_{\text{app}}\rangle$ and $\langle \mathcal{G};\Gamma_{\text{app}}\rangle$, but *not* the expansion-internal contexts, Δ and Γ. The final premise requires that the application site contexts are disjoint from the expansion-local type formation context. Because proto-expansions are ABTs identified up to alpha-equivalence, we can always discharge the final premise by alpha-varying the proto-expansion. This serves to enforce capture avoidance. Note that the purpose of $\mathbf{ML}^{\mathsf{Lit}}$ is to specify the necessary conditions, not to specify a particular implementation strategy. There are various ways to implement this capture avoidance condition. We will formally state the capture avoidance property in terms of capture avoiding substitution below, which is one strategy. Another is to pro-actively generate fresh variables for all internal bindings *a priori*.

The rule for references to spliced unexpanded types and patterns are analogous (see accompanying technical report). Note that we did not give examples of type splicing in the previous sections, but it is occasionally useful. For example, a TLM that implements a parser generator in the style of Menhir could need type splicing for reading in the non-terminal types, e.g. as on Line 9 of Fig. 6b.

5.5 Metatheory

Let us now sketch the main metatheoretic properties of $\mathbf{ML}^{\mathsf{Lit}}$.

5.5.1 Typed Expansion The first property that we are interested in is simple: that typed expansion produces a well-typed expansion. As it turns out, in order to prove this theorem, we must prove the following stronger theorem, because the proto-expression validation judgement is defined mutually inductively with the typed expansion judgement (due to splicing).

Theorem 5.1 (Typed Expression Expansion (Strong)).

(1) If $\langle \mathcal{D};\Delta\rangle\ \langle \mathcal{G};\Gamma\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e : \tau$ *then* $\Delta\ \Gamma \vdash e : \tau$.

(2) If $\Delta\ \Gamma \vdash^{\langle \mathcal{D};\Delta_{app}\rangle;\, \langle \mathcal{G};\Gamma_{app}\rangle;\, \hat{\Psi};\, \hat{\Phi};\, b} \grave{e} \leadsto e : \tau$ *and* $\Delta \cap \Delta_{app} = \emptyset$ *and* $dom(\Gamma) \cap dom(\Gamma_{app}) = \emptyset$ *then* $(\Delta \cup \Delta_{app})\ (\Gamma \cup \Gamma_{app}) \vdash e : \tau$.

The additional second clause simply states that the final expansion produced by proto-expression validation is well-typed under the combined application site and expansion-internal context (because spliced terms are distinguished only in the proto-expansion, but not in the final expansion). The proof proceeds by mutual rule induction and appeal to the analogous typed pattern expansion theorem and simple lemmas about type expansion and proto-type validation. The only issue is that it is not immediately clear that the mutual induction is well-founded, because the case in the proof of part 2 for Rule PEV-SPLICED invokes part 1 of the induction hypothesis on a term that is not a sub-term of the conclusion, but rather parsed out of the literal body, b. To establish that the mutual induction is well-founded, then, we need to explicitly establish a decreasing metric. The intuition is that parsing a term of out a literal body cannot produce a bigger term than the term that contained that very literal body. The details are given in the accompanying technical report.

5.5.2 TLM Reasoning Principles The following theorem summarizes the abstract reasoning principles that client programmers can rely on when applying a simple expression TLM. Informal descriptions of the labeled clauses are given inline, in gray boxes.

THEOREM 5.2 (SETLM REASONING PRINCIPLES). *If* $\langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}} \hat{a}$ ‘(b)‘ $\rightsquigarrow e : \tau$ *then*

1. ***(Expansion Typing)*** $\hat{\Psi} = \hat{\Psi}', \hat{a} \rightsquigarrow \_ \hookrightarrow \texttt{setlm}(\tau; e_{parse})$ *and* $\Delta\ \Gamma \vdash e : \tau$
   The type of the expansion is consistent with the type annotation on the applied seTLM definition.
2. ***(Responsibility)*** $b \downarrow_{\mathsf{Body}} e_{body}$ *and* $e_{parse}(e_{body}) \Downarrow \texttt{inj[SuccessE]}(e_{proto})$ *and* $e_{proto} \uparrow_{\mathsf{PrExpr}} \grave{e}$
   The parse function of the applied TLM is responsible for generating the proto-expansion.
3. ***(Segmentation)*** $\mathsf{seg}(\grave{e})$ segments b
   The segmentation determined by the proto-expansion segments the literal body.
4. ***(Segment Typing)*** $\mathsf{seg}(\grave{e}) = \{\texttt{splicedt}[m'_i; n'_i]\}_{0 \leq i < n_{ty}} \cup \{\texttt{splicede}[m_i; n_i; \grave{\tau}_i]\}_{0 \leq i < n_{exp}}$ *and*
   (a) $\{\langle \mathcal{D}; \Delta \rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow \tau'_i\ \mathsf{type}\}_{0 \leq i < n_{ty}}$ *and* $\{\Delta \vdash \tau'_i\ \mathsf{type}\}_{0 \leq i < n_{ty}}$
   Each spliced type has a well-formed expansion.
   (b) $\{\emptyset \vdash^{\langle \mathcal{D}; \Delta \rangle; b} \grave{\tau}_i \rightsquigarrow \tau_i\ \mathsf{type}\}_{0 \leq i < n_{exp}}$ *and* $\{\Delta \vdash \tau_i\ \mathsf{type}\}_{0 \leq i < n_{exp}}$
   Each segment type has a well-formed expansion.
   (c) $\{\langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}} \mathsf{parseUExp}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow e_i : \tau_i\}_{0 \leq i < n_{exp}}$ *and* $\{\Delta\ \Gamma \vdash e_i : \tau_i\}_{0 \leq i < n_{exp}}$
   Each spliced expression has a well-typed expansion consistent with the segment type.
5. ***(Capture Avoidance)*** $e = [\{\tau'_i/t_i\}_{0 \leq i < n_{ty}}, \{e_i/x_i\}_{0 \leq i < n_{exp}}]e'$ *for fresh variables* $\{x_i\}_{0 \leq i < n_{exp}}$ *and* $\{t_i\}_{0 \leq i < n_{ty}}$ *and some* e'
   We can decompose the final expansion, e, into an "internal expression", e′, with fresh variables in place of each spliced type or expression. The expansion can be produced by substituting in the expansions of these spliced types and expressions in the standard capture avoiding manner.
6. ***(Context Independence)*** $\mathsf{fv}(e') \subset \{t_i\}_{0 \leq i < n_{ty}} \cup \{x_i\}_{0 \leq i < n_{exp}}$
   The internal expression makes no mention of bindings in the application site context, i.e. the only free variables remaining are those standing for spliced terms.

Notice that we were able to state the hygiene properties (**Capture Avoidance** and **Context Independence**) without needing a notion of alpha-equivalence of source terms, as in typical formal accounts of hygiene [Adams 2015; Clinger and Rees 1991; Dybvig et al. 1992; Herman 2010; Herman and Wand 2008; Kohlbecker et al. 1986]. Instead, we used standard notions of capture avoiding substitution and free variables combined with the context disjointness conditions in the rules above. This is possible only because we keep track of spliced terms explicitly in the proto-expansion.

The reasoning principles theorem for pattern TLMs is below. The **Visibility** clause establishes that the hypotheses generated by a pattern of TLM application form are exactly the union of the hypothesis generated by the spliced patterns—there are no invisible bindings (see Sec. 3).

Theorem 5.3 (spTLM Abstract Reasoning Principles). *If* $\hat{\Delta} \vdash_{\hat{\Phi}} \hat{a}$ ‘(b)‘ $\rightsquigarrow p : \tau \dashv\Vdash \hat{\Gamma}$ *where* $\hat{\Delta} = \langle \mathcal{D}; \Delta \rangle$ *and* $\hat{\Gamma} = \langle \mathcal{G}; \Gamma \rangle$ *then all of the following hold:*

1. *(**Expansion Typing**)* $\hat{\Phi} = \hat{\Phi}', \hat{a} \rightsquigarrow \_ \hookrightarrow \mathtt{sptlm}(\tau; e_{parse})$ *and* $p : \tau \dashv\Vdash \Gamma$
2. *(**Responsibility**)* $b \downarrow_{\mathsf{Body}} e_{body}$ *and* $e_{parse}(e_{body}) \Downarrow \mathtt{inj[SuccessP]}(e_{proto})$ *and* $e_{proto} \uparrow_{\mathsf{PrPat}} \dot{p}$
3. *(**Segmentation**)* $\mathsf{seg}(\dot{p})$ segments b
4. *(**Segment Typing**)* $\mathsf{seg}(\dot{p}) = \{\mathtt{splicedt}[n'_i; m'_i]\}_{0 \leq i < n_{ty}} \cup \{\mathtt{splicedp}[m_i; n_i; \dot{\tau}_i]\}_{0 \leq i < n_{pat}}$ *and*
   (a) $\{\hat{\Delta} \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow \tau'_i \ \mathsf{type}\}_{0 \leq i < n_{ty}}$ *and* $\{\Delta \vdash \tau'_i \ \mathsf{type}\}_{0 \leq i < n_{ty}}$
   (b) $\{\emptyset \vdash^{\hat{\Delta}; b} \dot{\tau}_i \rightsquigarrow \tau_i \ \mathsf{type}\}_{0 \leq i < n_{pat}}$ *and* $\{\Delta \vdash \tau_i \ \mathsf{type}\}_{0 \leq i < n_{pat}}$
   (c) $\{\hat{\Delta} \vdash_{\hat{\Phi}} \mathsf{parseUPat}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow p_i : \tau_i \dashv\Vdash \langle \mathcal{G}_i; \Gamma_i \rangle\}_{0 \leq i < n_{pat}}$ *and* $\{p_i : \tau_i \dashv\Vdash \Gamma_i\}_{0 \leq i < n_{pat}}$
5. *(**Visibility**)* $\mathcal{G} = \biguplus_{0 \leq i < n_{pat}} \mathcal{G}_i$ *and* $\Gamma = \bigcup_{0 \leq i < n_{pat}} \Gamma_i$

6 IMPLEMENTATION

The implementation, `Relit`, which is an ongoing effort based on the finished design described in this paper, consists of a few changes to the context-free grammar of Reason together with a parse tree pre-processor for the OCaml compiler, currently using its PPX system (discussed in Sec. 7).

TLM Definitions The Reason grammar extension turns TLM definitions into module definitions of a stereotyped form. For example, the module `RegexNotation` defining `$regex` from Fig. 3b is, eliding a few minor details, shown below. The comments describe how the components correspond.

```
module RegexNotation = {
  module Notation_regex___relit /* = $regex */ = {
    type t = Regex.t; /* expansion type */
    module Lexer = RegexLexer; /* lexer module must match ocamllex sig */
    module Parser = RegexParser; /* parser module must match menhir sig */
    module Dependencies = { module Regex = Regex; } /* see text */
    exception Apply(string); /* used in TLM applications, see below */
  }
}
```

TLM definitions are constrained so that the signature of the generated module is a *singleton*, i.e. that it uniquely determines the module. This implies that the dependencies cannot list a value—only module paths and concrete types can be listed. As such, TLM definition equality is decidable and TLM definitions can be tracked in module types (signatures) without any special effort. Note that this is more restrictive than in the formal system in the previous section, where parse functions were included inline, rather than by path, and dependencies were arbitrary values. It must be possible to resolve the implementation of the lexer and parser modules, not just their signatures, at expansion time, so lexer and module paths are lifted to the top level of the compilation unit.

TLM Application For a TLM application, the Reason grammar extension produces an expression annotated with `relit` that raises the corresponding `Apply` exception, supplying a string containing the literal body. For example, a fragment of the example from Fig. 3c is below:

```
open RegexNotation; /* ... */
let restriction_template = (gene) => [%relit raise(Notation_regex___relit.Apply(
    "$(bisA)$(DNA.any_base)*$$(gene)$(DNA.any_base)*$(bisA)"))]
```

The Relit preprocessor rewrites TLM applications by following the specification in the previous sections differing only in that the “dependency variable” is a “dependency path”, here to `Notation_regex___relit.Dependencies`. The only difficult is in finding the TLM definition at all, which due to our integration into the module system may have been accessed indirectly, e.g. here via the **open** directive and in other situations via module synonyms, functors and so on. The trick is in the singleton restriction just described—if we can determine the signature for the module path that `Apply` is accessed from, we have the full definition in hand. To do so, we run the OCaml typechecker on this unexpanded representation. Because the TLM application raises an exception, its type can be generalized arbitrarily. As long as unrelated type errors at a particular depth are resolved (treating the TLM applications at that depth as "holes") expansion at that depth can proceed. This occasionally causes types that incidentally appear in error messages to be more general than the programmer might expect (e.g. here `restriction_template` will have type `'a => 'b` during this "pre-typing" phase). We are working to improve error reporting and, more generally, improving integration with various other tools in the Reason / OCaml ecosystem.

Proto-Expansion Validation The typing phase of proto-expansion validation also defers to the OCaml typechecker, with spliced expressions represented as variables of the corresponding segment type, resolved relative to the typing context determined at the call site. The recursively generated substitutions are, after validation is complete, substituted in the standard capture-avoiding manner.

OCaml Support Although our efforts are focused on Reason, it would not be difficult to add TLM-related forms to OCaml's standard grammar, and no changes to the preprocessor just described should be needed. The only caveat is that TLMs that support splicing would then, ideally, also be agnostic to the base language grammar in use. We are exploring various ways for TLM providers to opt in to this, e.g. by defining a version of `Relit.read_to` from Fig. 7 for both grammars.

7 RELATED WORK

In this section, we will give an overview of the many existing mechanisms that library providers might use to define new notation, and more specifically, new literal notation. Rather than evaluating these mechanisms by asking "how much syntactic power do they give the library provider?", however, our approach is to ask "what do these mechanisms quite reasonably **not** allow a library provider to express?", using the rubric of six reasoning principles that this paper has developed.

Syntax Definition Systems One approach available to library providers seeking to introduce new literal forms is to use a syntax definition system to extend the syntax of an existing language directly. There are hundreds of syntax definition systems of various design, and the parsers generated by these systems can be invoked to preprocess program text in various ways, e.g. by invoking them from within a build script, by using a preprocessor-aware compiler (e.g. `ocamlc`), or via language-integrated preprocessing directives, e.g. the import mechanism of SugarJ [Erdweg et al. 2011], or Racket's `#lang` directive [Flatt 2012].

These systems give a large amount of syntactic power to the notation provider, but this comes at a cost: with few exceptions, these systems make it difficult to reason abstractly about the six topics from Sec. 1. Rather than reiterating the points made there, let us focus on the exceptional cases.

A grammar extension system that has confronted the problem of **Responsibility** (but not the other problems) is Copper [Schwerdfeger 2010; Schwerdfeger and Van Wyk 2009]. Copper performs a modular grammar analysis that guarantees that determinism is conserved (i.e. ambiguities are not possible) when extensions of a certain restricted class are combined. The caveat is that the constituent extensions must prefix all newly introduced forms with marking tokens drawn from disjoint sets. To be confident that the marking tokens used are disjoint, providers must base them on, for example, the domain name system. Because the mechanism operates at the level of the context-free grammar, it is difficult for the client to define scoped abbreviations for these verbose marking tokens. TLMs can, in contrast, be abbreviated and distributed within modules following the usual scoping structure of the language. Composition is via splicing, rather than direct grammar composition.

Some programming languages, notably including theorem provers like Coq [Coq Development Team 2004] and Agda [Norell 2007], support "mixfix" notation directives [Griffin 1988; Missura 1997; Wieland 2009]. Many of these systems enforce **Capture Avoidance** and application-site **Context Independence** [Danielsson and Norell 2008; Griffin 1988; Coq Development Team 2004; Taha and Johann 2003]. The problem is that mixfix notation requires a fixed number of sub-trees, e.g. `if _ then _ else _`. Coq has some *ad hoc* extensions for list-like literals [Coq Development Team 2004]. These systems cannot express the example literal forms from this paper, because they can have any number of spliced terms.

Racket allows `#lang` definitions and reader macros to gain complete control over lexing and parsing the remainder of a file [Flatt 2012]. However, this flexibility makes it difficult to reason about segmentation and even responsibility ("is a reader macro active?") Providers can optionally generate fresh variables to avoid capture, but this is not enforced. We say more about macros below.

Lorenzen and Erdweg [2013, 2016]'s SoundExt is a grammar-based syntax extension system where extension providers can equip their new forms with derived typing rules. The system then attempts to automatically verify that the expansion logic, expressed using a rewrite system, rather than an arbitrary function, is sound relative to these derived rules, so it is possible to reason about **Expansion Typing**. SoundExt does not enforce hygiene, i.e. expansions might depend on the context and intentionally induce capture. A client can only indirectly reason about binding by inspecting the derived typing rules. There is no abstract segmentation discipline. Unlike TLMs, SoundExt supports type-dependent expansions [Lorenzen and Erdweg 2016]. The trade-off is that TLMs can generate proto-expansions, and therefore segmentations, even when the spliced expressions are ill-typed. Another important distinction is that TLMs rely on proto-expansion validation, rather than verification as in SoundExt (though providers can use MetaOCaml and related systems to reason about typing, as described in Sec. 2.2.4). The trade-off is that TLMs do not require a fully mechanized host language definition. Finally, there is no clear notion of parameterization or partial application in SoundExt or other syntax definition systems, so it would be difficult to define notation for, for example, all finite map implementations as we demonstrated in Sec. 4.

Term Rewriting Systems Another approach – and the approach that TLMs are rooted in – is to leave the context-free syntax of the language fixed, and instead contextually rewrite existing literal forms. For example, OCaml's textual syntax now includes *preprocessor extension (PPX) points* used to identify terms that some external term rewriter will rewrite [Leroy et al. 2014]. For example, we could mark a string literal as follows:

```
[%xml "<h1>Hello, {[first_name]}!</h1>"]
```

This does help with reasoning about responsibility, but technically more than one applied preprocessor might recognize this annotation (there are, in practice, many XML/HTML libraries), and annotations do not follow scoping rules. It is impossible to reason abstractly about the other issues because the code that the preprocessor generates is unconstrained.

Term-rewriting macro systems require that the client explicitly apply the intended rewriting, implemented by a scoped macro, to the term that is to be rewritten. This addresses the issue of **Responsibility** more thoroughly. However, unhygienic, untyped macro systems, like the earliest variants of the Lisp macro system [Hart 1963], and, more recently, Template Haskell [Sheard and Peyton Jones 2002] and GHC's quotation system [Mainland 2007] (which is based on Template Haskell), do not allow clients to reason abstractly about the remaining issues, again because the expansion that they produce is unconstrained. There is typically some way for library providers to opt in to a capture avoidance discipline by asking for fresh variables, but this is not enforced. Note that it is not enough that with Template Haskell / GHC quotation, the generated expansion is typechecked—to satisfy the **Expansion Typing** criterion, it must be possible to reason abstractly about *what the type of the generated expansion is*.

Hygienic macro systems prevent, or abstractly account for [Herman 2010; Herman and Wand 2008], **Capture**, and they enforce application-site **Context Independence** [Adams 2015; Clinger and Rees 1991; Dybvig et al. 1992; Kohlbecker et al. 1986]. The critical problem, detailed in Sec. 2.1.3, is that the standard context independence discipline makes it impossible to repurpose string literal forms to introduce compositional literal forms at other types.

integrate me

Hygienic term-rewriting macro systems, like those available in various Lisp-family languages [McCarthy 1978] and in Scala [Burmako 2013], cannot be used to repurpose string literals for composite literal notation at other types for exactly this reason. These systems would find that the appearance of `DNA.any_base` violates their context independence condition (which is one aspect of hygiene), because `DNA.any_base` is not, from the perspective of the context-free syntax, an argument to the macro nor even a sub-term of an argument for which a tree path can be assigned [Gorn 1965; Herman 2010; Herman and Wand 2008]. Instead, it arises as the result of performing a complex operation—parsing—on an arbitrary sub-sequence of the string literal passed into the macro. (It is possible for a macro to selectively opt out of the hygiene discipline—see Sec. 7 for more on why relying on this sort of "escape hatch" is an unreasonable solution.)

TLMs also consider the problem of definition-site context independence by specifying expansion dependencies explicitly. This sidesteps problems faced in prior attempts to integrate macros into module systems [Culpepper et al. 2005] related to "smuggling" definition site values to application sites, without violating the abstraction discipline of the module system. It also prevents issues related to cross-stage persistence, since macro definitions do not refer to definition-site values directly but rather program against the dependency signature.

Much of the research on macro systems has been for languages in the LISP tradition [McCarthy 1978] that do not have rich static type structure. The formal macro calculus studied by Herman and Wand [2008] (which is not capable of expressing new literal forms, for the reason just discussed) uses types only as a technical advice in reasoning about the binding structure of the generated expansion. The Scala macro system does support reasoning abstractly about **Expansion Typing** due to the return type annotations. The full calculus we have defined is the first detailed type-theoretic account of a typed, hygienic macro system of any design for an ML- or Scala-like language, i.e. one with a rich static type structure and support for pattern matching. Many of the basic mechanisms would be relevant even if support for parsing literal bodies was removed and replaced with term rewriting. Segmentations can be considered a refinement of the tree paths in prior work [Gorn 1965; Herman 2010].

Research on typed *staging macro systems* [Davies and Pfenning 1996] like MetaML [Sheard 1999], MetaOCaml [Kiselyov 2014], MacroML [Ganz et al. 2001; Taha and Johann 2003] and Typed Template Haskell [Jones [n. d.]] is not directly applicable to the problem of defining new literal forms because the syntax tree of the arguments cannot be inspected at all (staging macros are used mainly for optimization). Sec. 2.2.4 discussed how the typed quotations from these systems can help TLM providers reason about the type-correctness of their parsers. Future work could help address some of the mentioned limitations.

Some languages, including Scala [Odersky et al. 2008], build in *string splicing* (a.k.a. *string interpolation*) forms, or similar but more general *fragmentary quotation forms* [Slind 1991], e.g. SML/NJ [SML 2015]. These designate a particular delimiter to escape out into the expression language. The problem with using these together with macros as vehicles to introduce literal forms at various other types is 1) there is no "one-size-fits-all" escape delimiter, and 2) typing is problematic because every escaped term is checked against the same type. For example, in Fig. 1, we have splicing at two different types.

This brings us to the most closely related work, that of Omar et al. [2014] on *type-specific languages* (TSLs). Like simple expression TLMs (Sec. 2), TSLs allow library providers to programmatically control the parsing of expressions of generalized literal forms. With TSLs, parse functions are associated directly with nominal types and invoked according to a bidirectionally typed protocol. In contrast, TLMs are separately defined and explicitly applied. Accordingly, different TLMs can operate at the same type, and they can operate at any type, including structural types. In a subsequent short paper, Omar et al. [2015] suggested explicit application of simple expression TLMs also in a bidirectional typed setting [Pierce and Turner 2000], but this paper did not have any formal content. With TLMs, it is not necessary for the language to be bidirectionally typed (see Sec. 2.1.5 on ML-style type inference). The metatheory presented by Omar et al. [2014] establishes only that generated expansions are of the expected type (i.e. a variant of the Typed Expression Expansion theorem from Sec. 5.5), though the induction principle used in the proof is not clear. It does not establish the remaining abstract reasoning principles that have been the major focus of this paper. There appears to be a context independence condition but it does not allow for any dependencies whatsoever, which is unreasonably restrictive. There is no formal hygiene theorem and indeed the formal system in the paper does not correctly handle substitution or capture avoidance, issues we emphasized because they were non-obvious in Sec. 5. Moreover, the TLM does not guarantee that a valid segmentation will exist, nor associate types with spliced segments. Finally, this prior work did not consider pattern matching, module system integration or type- or module-parameterized type families. This paper addresses all of these, resulting in a design suitable for integration into Reason/OCaml.

8 DISCUSSION

The importance of specialized notation as a "tool for thought" has long been recognized [Iverson 1980]. According to Whitehead, a good notation "relieves the brain of unnecessary work" and "sets it free to concentrate on more advanced problems" [Cajori 1928], and indeed, advances in mathematics, science and programming have often been accompanied by new notation.

Of course, this desire to "relieve the brain of unnecessary work" has motivated not only the syntax but also the semantics of languages like ML and Scala – these languages maintain a strong type and binding discipline so that programmers, and their tools, can hold certain implementation details abstract when reasoning about program behavior. In the words of Reynolds [1983], "type structure is a syntactic discipline for enforcing levels of abstraction."

Previously, these two relief mechanisms were in tension—mechanisms that allowed programmers to express new notation would obscure the type and binding structure of the program text. TLMs resolve this tension for the broad class of literal forms that generalized literal forms subsume. This class includes all of the examples enumerated in Sec. 1 (up to the choice of outermost delimiter), the varied and non-trivial case studies outlined in this paper and in the accompanying technical report, and the examples collected from the empirical study by Omar et al. [2014]. We anticipate many more interesting examples will emerge as the Reason community explores the mechanism.

Of course, not all possible literal notation will prove to be in good taste. The reasoning principles that TLMs provide, which are the primary contributions of this paper, allow clients to "reason around" poor literal designs, using principles analogous to those already familiar to programmers in languages like ML and Scala. Although we emphasized integration with modules, our formal system demonstrates that modules are not necessary to capture the fundamental ideas in this paper. We intend this paper to be useful to the designers of languages far from the ML tradition. More generally, we intend for the "reasoning principles first" approach advocated in this paper to be a useful intellectual framework for language designers considering the merits of other "conveniences".

Limitations Not all interesting properties of a program will, in general, be apparent from its type, particularly in a language like OCaml. As usual, programmers will sometimes need to peek behind the abstraction boundaries or rely on informal documentation to understand, in detail, what a literal notation is doing (e.g. with respect its equational properties, its side effects and so on). TLMs in languages with more expressive type systems would be commensurately more reasonable.

Notation that intentionally introduces bindings into spliced terms, like Haskell's **do**-notation at types equipped with monadic structure, cannot be expressed using TLMs as defined in this paper, because splicing is capture avoiding. Although we considered weakening the capture avoidance condition in various somewhat reasonable ways, e.g. by communicating which identifiers are explicitly captured by each spliced segment, it is quite difficult to communicate this structure to client programmers even with tool support. Given that **do**-notation is already general and comes equipped with well-understood reasoning principles, the most reasonable approach is perhaps to build it in primitively, suitably parameterized over the implementation of the `MONAD` signature.

Another future direction has to do with automated refactoring. The unexpanded language does not come with context-free notions of renaming and substitution. However, given a segmentation, it should be possible to "backpatch" refactorings into literal bodies. Recent work by Pombrio et al. [2017] on tracking bindings "backwards" from an expansion to the source program is likely relevant. The challenge is that the TLM's splicing logic might not be invariant to refactorings.

At several points in the paper, we allude to editor integration. However, several important questions having to do with TLM-specific syntax highlighting, incremental parsing and error recovery [Graham et al. 1979] remain to be considered. Another interesting direction would be to generalize TLMs to support non-textual notation in the setting of a structure editor [Omar et al. 2017].

Concluding Remarks To conclude, let us briefly reiterate the key ideas of this paper. The focus was on ensuring that client programmers can follow simple protocols when they have questions about the syntactic structure, type structure or binding structure of a program. In answering these sorts of questions, the client is not made to look at the generated expansion itself, nor inspect the parser implementation. Instead, the programmer need only be given knowledge of the expansion type from the TLM definition and the segmentation inferred by the expander at each application site, which carries a small volume of information that can be communicated straightforwardly by standard editor services. Certain questions related to the binding structure simply do not need to be asked due to the strict context independence and capture avoidance discipline of the system, enabled by the explicit tracking of dependencies, and of spliced segments.

Despite these semantic constraints, the system is able to express a number of non-trivial examples with few compromises because there is only one simple lexical constraint on literal bodies. The mechanism integrates cleanly into Reason/OCaml, with full support for its type system, module system and pattern matching system. TLM providers can use existing, mature parser generators and parse tree representations, and can opt-in to a stronger typing discipline by using various implementations of MetaOCaml. Overall, we believe that TLMs represent a distinctly *reasonable* and *expressive* new point in the design space of literal notation definition systems.

ACKNOWLEDGMENTS

This work was supported in part by AFRL and DARPA under agreement #FA8750-16-2-0042; by the NSA under lablet contract #H98230-14-C-0140; and by Ravi Chugh at the University of Chicago. The authors would like to thank Robert Harper, Karl Crary and Eric Van Wyk for valuable feedback on the first author's doctoral dissertation [Omar 2017], which forms the basis of this work; Charles Chamberlain for his crucial contributions to the implementation and for valuable feedback on the paper; and members of the PL Reading Group at University of Chicago and the anonymous referees for their thoughtful critiques, which substantially improved the paper.

REFERENCES

2015. SML/NJ Quote/Antiquote. http://www.smlnj.org/doc/quote.html. Retrieved Nov. 3, 2015.

Michael D. Adams. 2015. Towards the Essence of Hygiene. In *POPL*. http://doi.acm.org/10.1145/2676726.2677013

Eric Anderson, Gilman D Veith, and David Weininger. 1987. *SMILES, a line notation and computerized interpreter for chemical structures*. US Environmental Protection Agency, Environmental Research Laboratory.

Anton Bachin. 2018. Markup.ml — Error-recovering streaming HTML5 and XML parsers for OCaml. http://aantron.github.io/markup.ml/. Retrieved Mar. 14, 2018.

Alan Bawden. 1999. Quasiquotation in Lisp. In *Partial Evaluation and Semantic-Based Program Manipulation*. http://repository.readscheme.org/ftp/papers/pepm99/bawden.pdf

Frédéric Bour, Thomas Refis, and Gabriel Scherer. 2018. Experience report: Merlin, a Language Server for OCaml. http://gallium.inria.fr/~scherer/drafts/merlin.pdf. Retrieved June 21, 2018. In *Conditionally accepted at ICFP 2018*.

Martin Bravenboer, Eelco Dolstra, and Eelco Visser. 2007. Preventing Injection Attacks with Syntax Embeddings. In *GPCE*. http://doi.acm.org/10.1145/1289971.1289975

Eugene Burmako. 2013. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *4th Workshop on Scala*. Article 3, 10 pages.

Florian Cajori. 1928. *A history of mathematical notations*. Vol. 1. Courier Corporation.

Adam Chlipala. 2010. Ur: statically-typed metaprogramming with type-level record computation. In *PLDI*. http://doi.acm.org/10.1145/1806596.1806612

Adam Chlipala. 2015. Ur/Web: A Simple Model for Programming the Web. In *POPL*. http://dl.acm.org/citation.cfm?id=2676726

William D. Clinger and Jonathan Rees. 1991. Macros That Work. In *POPL*. http://doi.acm.org/10.1145/99583.99607

Ryan Culpepper, Scott Owens, and Matthew Flatt. 2005. Syntactic abstraction in component interfaces. In *International Conference on Generative Programming and Component Engineering (GPCE)*.

Nils Anders Danielsson and Ulf Norell. 2008. Parsing Mixfix Operators. In *20th International Symposium on Implementation and Application of Functional Languages (IFL) - Revised Selected Papers*. http://dx.doi.org/10.1007/978-3-642-24452-0_5

Rowan Davies and Frank Pfenning. 1996. A Modal Analysis of Staged Computation. In *POPL*.

Daniel de Rauglaudre. 2003. Camlp4 reference manual. *Online (September 2003)* (2003).

Stephen Dolan. 2018. Staged Metaprogramming in stock OCaml. https://github.com/stedolan/ppx_stage/. Retrieved Mar. 14, 2018.

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. 1992. Syntactic Abstraction in Scheme. *Lisp and Symbolic Computation* 5, 4 (1992), 295–326.

Sebastian Erdweg, Tillmann Rendel, Christian Kastner, and Klaus Ostermann. 2011. SugarJ: Library-based syntactic language extensibility. In *OOPSLA*.

Sebastian Erdweg and Felix Rieger. 2013. A framework for extensible languages. In *GPCE*.

Sebastian Erdweg, Felix Rieger, Tillmann Rendel, and Klaus Ostermann. 2012. Layout-sensitive language extensibility with SugarHaskell. In *Proceedings of the 2012 Symposium on Haskell*. ACM, 149–160.

Matthew Flatt. 2012. Creating Languages in Racket. *Commun. ACM* 55, 1 (Jan. 2012), 48–56. http://doi.acm.org/10.1145/2063176.2063195

Alain Frisch, Peter Zotov, and Gabriel Radanne. 2017. ppx_tools: Tools for authors of ppx rewriters. https://github.com/ocaml-ppx/ppx_tools. Retrieved June 22, 2018.

Steven Ganz, Amr Sabry, and Walid Taha. 2001. Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML. In *ICFP*.

Saul Gorn. 1965. Explicit definitions and linguistic dominoes. In *Systems and Computer Science, Proceedings of the Conference held at Univ. of Western Ontario*. 77–115.

Susan L Graham, Charles B Haley, and William N Joy. 1979. *Practical LR Error Recovery*. Vol. 14. ACM.

T. R. G. Green. 1989. Cognitive Dimensions of Notations. In *Proceedings of the HCI'89 Conference on People and Computers V (Cognitive Ergonomics)*. 443–460.

T.G. Griffin. 1988. Notational definition-a formal account. In *Logic in Computer Science (LICS '88)*. 372–383.

Robert Harper. 1997. Programming in Standard ML. http://www.cs.cmu.edu/~rwh/smlbook/book.pdf. Working draft, retrieved June 21, 2015.

Robert Harper. 2012. *Practical Foundations for Programming Languages*. Cambridge University Press.

T. P. Hart. 1963. *MACRO Definitions for LISP*. Report A. I. MEMO 57. Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts.

David Herman. 2010. *A Theory of Typed Hygienic Macros*. Ph.D. Dissertation. Northeastern University, Boston, MA.

David Herman and Mitchell Wand. 2008. A Theory of Hygienic Macros. In *ESOP*.

Graham Hutton. 1992. Higher-order functions for parsing. *Journal of Functional Programming* 2, 3 (July 1992), 323–343. http://www.cs.nott.ac.uk/Department/Staff/gmh/parsing.ps

IEEE. 2016. Information Technology Portable Operating System Interface (POSIX) Base Specifications, Issue 7. (2016).

Kenneth E. Iverson. 1980. Notation as a Tool of Thought. *Commun. ACM* 23, 8 (1980), 444–465. https://doi.org/10.1145/358896.358899

Stephen C Johnson. 1975. *Yacc: Yet another compiler-compiler*. Vol. 32. Bell Laboratories Murray Hill, NJ.

Simon Peyton Jones. [n. d.]. New directions for Template Haskell.

Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR (1) parsers. In *European Symposium on Programming*. Springer, 397–416.

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml - System Description. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*. 86–102. https://doi.org/10.1007/978-3-319-07151-0_6

Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic Macro Expansion. In *Symposium on LISP and Functional Programming*. 151–161.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2014. *The OCaml system release 4.02 Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique.

Barbara Liskov and Stephen Zilles. 1974. Programming with abstract data types. In *ACM SIGPLAN Notices*, Vol. 9. ACM, 50–59.

Florian Lorenzen and Sebastian Erdweg. 2013. Modular and automated type-soundness verification for language extensions. In *ICFP*. 331–342. http://dl.acm.org/citation.cfm?id=2500365

Florian Lorenzen and Sebastian Erdweg. 2016. Sound type-dependent syntactic language extension. In *POPL*. http://dl.acm.org/citation.cfm?id=2837614

Geoffrey Mainland. 2007. Why it's nice to be quoted: quasiquoting for Haskell. In *Haskell Workshop*.

Coq Development Team. 2004. *The Coq proof assistant reference manual*. LogiCal Project. http://coq.inria.fr Version 8.0.

J. McCarthy. 1978. History of LISP. In *History of programming languages I*. ACM, 173–185.

Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: reconciling object, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. ACM.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The Definition of Standard ML (Revised)*. The MIT Press.

Stephan Albert Missura. 1997. *Higher-Order Mixfix Syntax for Representing Mathematical Notation and its Parsing*. Ph.D. Dissertation. ETH Zurich.

John C Mitchell and Gordon D Plotkin. 1988. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 10, 3 (1988), 470–502.

U. Norell. 2007. Towards a practical programming language based on dependent type theory. *PhD thesis, Chalmers University of Technology* (2007).

Martin Odersky, Lex Spoon, and Bill Venners. 2008. *Programming in Scala*. Artima Inc.

Cyrus Omar. 2017. *Reasonably Programmable Syntax*. Ph.D. Dissertation. Carnegie Mellon University.

Cyrus Omar and Jonathan Aldrich. 2018. *Reasonably Programmable Literal Notation: Supplemental Material*. Technical Report CMU-ISR-18-104. Carnegie Mellon University.

Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. 2014. Safely Composable Type-Specific Languages. In *ECOOP*.

Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: a bidirectionally typed structure editor calculus. In *POPL*. http://dl.acm.org/citation.cfm?id=3009900

Cyrus Omar, Chenglong Wang, and Jonathan Aldrich. 2015. Composable and Hygienic Typed Syntax Macros. In *ACM Symposium on Applied Computing (SAC)*.

OWASP. 2017. OWASP Top 10 2013. https://www.owasp.org/index.php/Top_10_2017-Top_10. Retrieved May 28, 2017.

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 1–44. http://doi.acm.org/10.1145/345099.345100

Justin Pombrio, Shriram Krishnamurthi, and Mitchell Wand. 2017. Inferring Scope through Syntactic Sugar. (2017).

François Pottier and Yann Régis-Gianas. 2016. The Menhir parser generator. *See: http://gallium. inria. fr/fpottier/menhir* (2016).

Reason Team. 2018. Reason Guide: What and Why? https://reasonml.github.io/docs/en/what-and-why.html. Retrieved Mar. 14, 2018.

J C Reynolds. 1983. Types, abstraction and parametric polymorphism. In *IFIP Congress*.

August Schwerdfeger. 2010. *Context-aware scanning and determinism-preserving grammar composition, in theory and practice*. Ph.D. Dissertation. University of Minnesota.

August Schwerdfeger and Eric Van Wyk. 2009. Verifiable composition of deterministic grammars. In *PLDI*. http://doi.acm.org/10.1145/1542476.1542499

Denys Shabalin, Eugene Burmako, and Martin Odersky. 2013. *Quasiquotes for Scala*. Technical Report EPFL-REPORT-185242.

Tim Sheard. 1999. Using MetaML: A Staged Programming Language. *Lecture Notes in Computer Science* 1608 (1999).

Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Haskell Workshop*.

Konrad Slind. 1991. Object language embedding in Standard ML of New-Jersey. In *ICFP*.

Walid Taha. 2004. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*. Springer, 30–50.

Walid Taha and Patricia Johann. 2003. Staged Notational Definitions. In *GPCE*. http://dx.doi.org/10.1007/978-3-540-39815-8_6

D. R. Tarditi and A. W. Appel. 1990. *ML-Yacc, Version 2.0 Documentation*.

Ken Thompson. 1968. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (June 1968), 419–422. http://doi.acm.org/10.1145/363347.363387

Arie van Deursen, Paul Klint, and Frank Tip. 1993. Origin Tracking. *J. Symb. Comput.* (1993), 523–545. http://dx.doi.org/10.1016/S0747-7171(06)80004-0

Arthur Whitney and Dennis Shasha. 2001. Lots O'Ticks: Real Time High Performance Time Series Queries on Billions of Trades and Quotes. *SIGMOD Rec.* 30, 2 (May 2001), 617–617. https://doi.org/10.1145/376284.375783

Jacob Wieland. 2009. *Parsing Mixfix Expressions*. Ph.D. Dissertation. Technische Universitat Berlin.