Reasonably Programmable Literal Notation

ANONYMOUS AUTHOR(S)

General-purpose programming languages typically define literal notation for only a small number of common data structures, e.g. lists. This is unsatisfying because there are many other data structures for which literal notation might be useful, e.g. hash tables, regular expressions, HTML data, SQL queries, syntax trees and chemical structures. There may also be different implementations of each of these data structures, e.g. with different performance characteristics, 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 providers can reason modularly about syntactic ambiguity, and TLM clients can reason abstractly, i.e. without examining the underlying expansion, about types and binding. The system only needs to convey to clients, via secondary notation, the inferred *segmentation* of each literal, which gives the locations and types of spliced subterms. This paper establishes these abstract reasoning principles formally with a calculus of typed expressions, pattern matching and ML-style modules. This calculus is the first detailed type-theoretic account of a hygienic macro system, of any design, for a language with these essential features of ML. We are integrating TLMs into Reason, an emerging alternative front-end for OCaml.

1 Introduction

When designing the surface syntax of a general-purpose programming language, it is common practice to define *literal notation* that decreases the syntactic cost of constructing and pattern matching over values of a select few data structures. For example, many languages in the ML family support list literals like `[x1, x2, x3]` in both expression and pattern position [25, 48]. While lists are common across problem domains, other literal notation is more specialized. For example, Ur/Web extends the surface syntax of Ur (an ML-like language [9]) with expression and pattern literals for encodings of XML and HTML data [10]. The example in Fig. 1 below shows two HTML literals, one that "splices in" a string expression delimited by `{[` and `]}` (Line 1) and the other an HTML expression delimited by `{` and `}` (Line 2).

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

Fig. 1. HTML literals with support for splicing at two different types are built primitively into Ur/Web [10].

This design practice, where the language designer privileges certain library constructs with built-in literal notation, is decidedly *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 [8, 32, 53]. For example, 1) clients of a "collections" library might want not just list literals, but also literal notation for matrices, finite sets, maps and so on; 2) clients of a "web programming" library might like CSS literals (which Ur/Web lacks); 3) a compiler author might like "quotation" literals for the terms of the object language and various intermediate languages of interest; and 4) clients of a "chemistry" library might like chemical structure literals based on the SMILES standard [4].

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. [6] finds that literal notation for structured encodings of queries, like the SQL-based query literals now found in many languages [46], reduce the temptation to use string encodings of queries and therefore reduce the risk of catastrophic string injection attacks [55]. 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. [53] sampled strings from open source projects and found

2017. 2475-1421/2017/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 w = compute_w();
8 let x = compute_x(w);
9 let y = `((!R)@&{&/x!/:2_!x}'!R)`;
```

(a) It is difficult to reason abstractly about programs that use unfamiliar grammar extensions (see the five reasoning criteria in the text).

THIS PAPER: TYPED LITERAL MACROS

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

(b) TLMs make examples like these more reasonable by leaving the base grammar fixed and making the type and binding structure and the segmentation explicit.

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

that at least 15% of them could be parsed by some readily apparent type-specific grammar. Literal notation, with support for splicing, would decrease the syntactic cost of composite encodings of this data, which is more amenable to programmatic manipulation and compositional reasoning.

Of course, it would not be scalable 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 [37] and Sugar* [17, 18], macro-based systems like Omar et al's *type-specific languages* (TSLs) [53], and other systems that we will discuss in Sec. 8 can all be used to define new literal notation (and, in some cases, other forms of new notation, such as new infix expression forms 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 syntax extension systems make it difficult or impossible to reason abstractly about such fundamental issues as types and variable binding. 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 example, the perspective of a programmer attempting to comprehend the program text in Figure 2a, which is written in a dialect of OCaml's surface syntax called Reason [66] that has, hypothetically, been extended with some number of new literal forms by a grammar extension system – Lines 1-5 show the grammar extension syntax from Camlp4 [37], but Sugar*/SugarHaskell is similar [17–19]. The literal form on Line 9 constructs an encoding of a query in the niche stack-based database query language K, using its intentionally terse notation [70]. The problem is that a programmer looking at the program as presented, and unfamiliar with (i.e. holding abstract) the details elided on Lines 5-6, cannot easily answer questions like the following:

- ○ **Responsibility**: Which syntax extension even determines the expansion of the literal on Line 9? Might activating a new extension generate a conflicting expansion for the same form?
- ○ **Segmentation**: Are the characters `x`, `R` and `2` on Line 9 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 form (e.g. as operators in the K query language)?
- ○ **Capture**: If `x` is in fact a spliced expression, does it refer to the binding of `x` on Line 5, or might it capture an unseen binding of the same identifier in the expansion of Line 9?
- ○ **Context Dependence**: If `w`, on Lines 7-8, is renamed, could that break Line 9 because the expansion assumes that `w` is in scope? What other hidden assumptions is the expansion making?
- ○ **Typing**: What type does `y` have? What type is each spliced expression expected to have?

Forcing the programmer to reason transparently to answer basic questions like these defeats the ultimate purpose of syntactic sugar: decreasing cognitive cost [23]. Analagous 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 [59]. Ideally, the programmer would be able to abstract in some analagous manner over the implementation of an unfamiliar notation.

An approach that has made some headway toward this ideal is Omar et al.'s *type-specific languages* (TSLs) [53]. A language that supports TSLs delegates control over the parsing and expansion of certain *generalized literal forms* to a parser associated with the type that the form is being checked against. So if we add TSLs to our language, our example then requires a type annotation on `y`:

```
let y : KQuery.t = `((!R)@&{&/x!/:2_!x}'!R)`
```

The parser associated with the type `KQuery.t` is statically invoked to lex, parse and expand the literal body, i.e. the sequence of characters between the outer delimiters, here `` `( `` and `` )` ``. The literal body is constrained by the context-free grammar of the language only in that nested delimiters must be matched (like comments in OCaml). TSLs are closely related to the mechanism that we will propose, so let us analyze TSLs from the perspective of the five reasoning criteria just outlined.

- ◐ **Responsibility**: The type-directed dispatch mechanism allows the client to easily determine which parser is responsible for each literal form: the parser that was associated with the type when it was defined. Clients do not need to worry about different TSLs conflicting syntactically because the context-free grammar of the language remains fixed and composition is mediated by splicing. The main limitation here is that it is impossible to define literal notation after a type has been defined, and it is also impossible to define multiple notations at a single type.
- ◐ **Segmentation**: The parser can splice expressions out of the literal body, but there is no clear way to associate each spliced term with some particular segment (i.e. subsequence) of the literal body, nor is there a guarantee that spliced terms are non-overlapping. As such, there is no way to indicate to the programmer where base language expressions are located within a literal form.
- ◐ **Capture**: The TSL mechanism as described enforces complete capture avoidance (though the formal specification given in the paper did not adequately enforce this constraint).
- ◐ **Context Dependence**: The TSL mechanism as described requires that the expansion be completely closed, so it is trivially context independent. However, this comes at a significant expressive cost: the generated expansions cannot make use of any libraries whatsoever.
- ◐ **Typing**: The expansion must necessarily be of the associated type, so abstract reasoning about the type of the expansion is straightforward. However, there is no easy way to determine the type expected for each spliced expression. In addition, the prior work considered only a monomorphic, nominally-typed language with local type inference, leaving open a number of problems that came up as we considered integrating TSLs into Reason/OCaml:
  (a) **Structural Types**: There is no way to define literal notation at structural types, e.g. tuple and arrow types, because there is no "definition site" for such types.
  (b) **Parameterized & Abstract Types**: There is no way to define literal notation over a type-parameterized family of types, e.g. at all types `list('a)`. Similarly, there is no way to define literal notation over a module-parameterized family of abstract types, e.g. at every abstract type defined by a module implementing the `QUEUE` signature. Parameterized and abstract type families are ubiquitous in ML-family languages.
  (c) **Pattern Literals**: Pattern matching is ubiquitous in ML-family languages but the prior work on TSLs considered only expression literals.
  (d) **ML-Style Type Inference**: It is not clear that a type-directed dispatch scheme can be cleanly reconciled with ML-style type inference.

We consider other existing approaches against these criteria in Sec. 8. Taken together, we found that no existing approach was reasonable and expressive enough for integration into Reason/OCaml.

Contributions This paper introduces *typed literal macros* (TLMs): the first system for defining new literal notation that (1) provably maintains all five of the abstract reasoning principles 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 (1) a series of increasingly ML-like calculi equipped with proofs of these abstract reasoning principles; and (2) a number of non-trivial examples that appear throughout the paper and involve the language features mentioned above. In describing these examples, we also 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 client-side reasoning principles.

TLMs, like TSLs, operate on generalized literal forms. With TLMs, however, the mechanism for defining new literal notation is not tied up with the type definition mechanism of the language. TLMs are instead explicitly named and applied. For example, Line 9 of Figure 2b applies a TLM named `$kq` to a generalized literal form delimited by `` `( `` and `` )` `` to express the K query example. The programmer can reason abstractly as follows:

- **Responsibility**: The lexer and parser specified by the applied TLM on Lines 2-3 is exclusively responsible for lexing, parsing and expanding the literal body. We will give more details on constructing a TLM lexer and parser in the next section. Due to this explicit dispatch mechanism, there can be any number of TLMs defined at a given type. As with TSLs, the context-free grammar of the language is fixed, so the TLM provider can modularly establish that the notation that the TLM implements is unambiguous (even in situations where a spliced term itself applies another TLM, which we will show in a later example).
- **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 from segments that are parsed in some other way by the TLM. The segmentation is all that needs to be communicated to editor services downstream of the expander, and ultimately to the programmer using secondary notation, e.g. colors in this document. So by examining Figure 2b, the programmer knows that the two instances of `x` on Line 3 are parsed as spliced expressions (because they are black), whereas the `R`'s must be parsed in some other way, e.g. as operators in the K language (because they are green).
- **Capture**: Splicing is capture-avoiding, so the spliced expression `x` must refer to the binding of `x` on Line 2. It cannot capture a coincidental binding of `x` in the expansion. We will say more about capture later on.
- **Context Dependence**: The system enforces context independence, so the expansion of Line 3 cannot rely on the fact that, e.g., `w` is in scope. The client is free to rename `w`. The external dependencies of the generated expansion are explicit – here, Lines 4-5 specify that expansions generated by `$kq` require the module `KQuery` – and the system ensures that this dependency is bound as specified even if the identifier `KQuery` has been shadowed at the application site.
- **Typing**: The type annotation on the definition of `$kq` (Line 1) determines the type that the expansion is validated against, here `KQuery.t` (not shown). Moreover, each spliced segment in the inferred segmentation also has a type annotation. This, together with the context independence property, ensures that type inference at the TLM application site can be performed (by editor services or in the programmer's mind) abstractly, i.e. given only the inferred segmentation rather than the full expansion. We will see examples of TLMs that make use of more advanced features of the OCaml type system – notably, pattern matching, parameterized types and modules – later.

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

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

```
notation $regex at Regex.t
  lexer    RegexLexer
  parser   RegexParser.start
  expansions require
    module Regex as Regex;
```

(b) The definition of the `$regex` TLM. Figure **??** defines `RegexLexer` and `RegexParser`.

```
1 let any_base = $regex `(A|T|G|C)`;
2 let bisA = $regex `(GC$(any_base)GC)`;
3 let gene_restriction_template(gene) =>
4   $regex `($(bisA)$(any_base)*$$(gene)$(any_base)*$(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.

Outline Sec. 2 introduces expression TLMs with a detailed case study. Sec. 3 then introduces pattern TLMs, i.e. TLMs that generate patterns, and describes the special reasoning conditions in pattern position. Sec. 4 then introduces the more general *parametric TLMs*, i.e. TLMs that can take type and module parameters, which allow us to define literal notation over a type- or module-parameterized family of types. This section also reveals that the **`expansions require`** clause is but syntactic sugar for parameterization followed immediately by partial parameter application. Having introduced the basic machinery, we continue in Sec. 5 with examples that further demonstrate the expressive power of this approach and various parser implementation strategies that notation providers can consider. This notably includes our take on the example from Fig. 1 of Ur/Web-style HTML literals. Next, Sec. 6 defines a type-theoretic calculus of simple expression and pattern TLMs and formally establishes the reasoning principles implied above in their simplest form. Sec. 7 adds type functions and an ML-style module system to this calculus, and gives a more general variant of the reasoning principles theorem. Sec. 8 compares TLMs to related work, guided by the rubric of five reasoning principles just discussed. Sec. 9 concludes with a summary of the contributions of this paper and a discussion of future work. Certain technical details, proofs and extensions to the calculi presented in this paper are given in the supplement.

2 Expression TLMs

Consider the recursive datatype `Regex.t` defined by the module in Fig. 3a, which encodes *regular expressions* (regexes) into Reason [67]. Regular expressions are common in fields like 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")))))
```

In Reason, the notation `Regex.(`e`)` locally opens the module `Regex` for use within the expression e, so we do not need to qualify the constructors `Regex.Or` and `Regex.Str`. Even allowing for this shorthand, however, this notation for constructing regexes is rather unwieldy. Instead, we would like to be able to use the common POSIX-style notation for constructing values of type `Regex.t` [1].

2.1 TLM Definition and Application

Figure 3b defines a TLM named `$regex` for values of type `Regex.t` that supports a POSIX-style literal notation extended with splice forms for regexes and strings. Line 1 of Fig. 3c applies this TLM to construct the regex `any_base` just described. Line 2 of Fig. 3c then applies `$regex` again, using its regex splice form to construct a regex `bisA` matching the DNA sequences recognized by the BisA restriction enzyme, where the middle base can be any base. Finally, Lines 3-4 of Fig. 3c define a function that constructs a more complex regex from these two regexes and a given gene sequence represented as a string, which is spliced in using a string splice form delimited by `$$(` and `)`.

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

```
$regex `(GC$(any_base)GC)`
```

According to the context-free grammar of the language, this form is simply a leaf of the unexpanded syntax tree, like a string literal would be. TLM names are prefixed by `$` to distinguish them from variables. There are no lexical constraints on the body of the generalized literal form except that any occurrences of `` `( `` must be balanced by `` )` ``, much like nested comments in OCaml, Reason and other languages. Generalized literal forms, which first arose in the prior work on TSLs [53], therefore syntactically subsume other literal forms. The prior work specified several other choices of outer delimitation, including layout-sensitive delimitation, but for the purposes of this work, we will use `` `( `` and `` )` `` exclusively. The fact that the context-free grammar of the base language is fixed ensures that notation providers can reason modularly about syntactic ambiguity in their own grammars.

Notice that in the unexpanded expression above, we did not distinguish the spliced expression from the rest of the literal. This is to emphasize that it is only during the subsequent *expansion* phase that the body of the generalized literal form is lexed, parsed and expanded to produce an expanded syntax tree where spliced expressions, like `any_base`, have been revealed. Responsibility for lexing, parsing and expanding the literal body is delegated to the lexer and parser specified by the applied TLM by the clauses **`lexer`** and **`parser`**, respectively.

2.2 Abstract Reasoning Principles

Rather than looking immediately at Fig. **??**, which defines the lexer and parser specified by `$regex`, let us first consider the perspective of a client programmer who does not want to delve into the details of lexing and parsing for each piece of literal notation that they encounter. What can this programmer deduce just by reasoning from the information presented in Figures 3a through 3c?

From 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 1 of Fig. 3b. This is analagous to the return type of a function. The identity of `Regex.t` is determined relative to the TLM definition site, not at each application site (so the module `Regex` need not be in scope at the application site, or it can be shadowed by a different module).

The TLM mechanism enforces an similarly strong context independence property on generated expansions by requiring that the TLM explicitly specify the modules that the expansions it generates might internally depend on. In this case, Lines 4-5 of Fig. 3b specify that generated expansion might use the module `Regex`, again as it is bound at the definition site, using the module variable `Regex` internally. These happen to coincide in this case, but in general, the dependency can be an arbitrary module path while internally it must be bound to a module variable, for example:

```
expansions require
  module Regex as Regex
  module Core.List as L
```

The module variable `L` will be bound internally to `Core.List` in all expansions generated by this TLM, including those where `L` or `Core.List` are not bound, or otherwise bound. All other bindings, whether at the TLM definition site or the TLM application site, are not internally available to the expansion. This allows client programmers to freely rename and shadow variables as they are used to, without needing to inspect each parser to determine whether it requires certain modules be bound to particular names at each use site (as is common with, e.g., syntax extensions based on `camlp4` or Sugar*, or when using unhygienic term rewriting approaches like OCaml's PPX system or Template Haskell [62], as further discussed in Sec. 8).

Enforcing this strong context independence condition is quite subtle because TLM parsers need to be able to extract spliced expressions from the literal body and place them in the final expansion. Naïvely checking that the final expansion is closed except for the explicitly named dependencies would inappropriately enforce context independence on application site spliced expressions, which should certainly not be prevented from referring to variables in the application site context. For example, consider the final expansion of the example from Line 2 of Fig. 3c above:

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

In this term, both `Regex` and `any_base` are free variables. There is nothing to indicate which free variables came from the literal body via an intentional act of splicing by the TLM. As we will discuss more extensively in Sec. 8, this is why traditional hygienic term-rewriting macro systems, like those available in various Lisp-family languages [45] and in Scala [7], cannot be used to repurpose string literals for literal notation at other types.

To address this problem, the TLM parser does not generate the final expansion directly. Instead, the parser generates a *proto-expansion* that refers to spliced expressions 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 expressed as follows:

```
Regex.Seq(Regex.Str "GC", Regex.Seq(spliced<4; 11; Regex.t>, Regex.Str "GC"))
```

Here, `spliced<4; 11; Regex.t>` is a reference to the spliced expression `any_base` by its location relative to the start of the literal body being expanded. The context independence condition can be enforced straightforwardly on the proto-expansion – the only free variable in the proto-expansion is `Regex`, which is an explicitly listed dependency.

The set of splice references in the proto-expansion generated for a literal body is called the *segmentation* of that literal body. The system checks that the segmentation does indeed segment the literal body, i.e. that the segments are non-overlapping. For the purposes of abstract reasoning, only the segmentation needs to be reported to downstream editor services, and ultimately to the programmer. The rest of the proto-expansion can be held abstract. In this paper, spliced segment locations are reported using colors, with segments that are not part of spliced segment shown in color (green, so far) while spliced segments start out black. When TLM applications are nested, a distinct color can be used at each depth. For example, in Sec. 5, we will describe a TLM `$html` for HTML notation in the style of Ur/Web (see Fig. 1). We might apply it, together with another TLM `$smiles` for chemical structures described using the SMILES standard [4], as follows:

```
$html `(<div>Chemical structure of sucrose: {
  Smiles.to_svg
    $smiles `({mono_glucose}-O-{mono_fructose})`
}</div>)`
```

Each splice reference in the segmentation gives not just the location of the spliced segment but also a type annotation. Returning to the regex example above, the specified type annotation is `Regex.t`. This type annotation, which must also be context independent, constrains the spliced expression. Consequently, type inference can be performed using only the segmentation together with the type annotation on the applied TLM's definition. For example, consider the function `gene_restriction_template` on Lines 3-4 of Fig. 3c. The return type of this function can be determined to be `Regex.t` because of the type annotation on the `$regex` TLM. 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. Context independence 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 expansion held abstract. Spliced segment types can be communicated directly to the programmer upon request by an editor service, e.g. `merlin` for OCaml/Reason [2].

————————- stuff below is notes / not yet revised

When a TLM definition appears inside a module, it must also appear in the signature with the same specification, up to the usual notions of type and module path equivalence in the type annotation and module dependencies (like datatype definitions in ML). Note that even when a TLM needs only a single helper function, it must be placed into a named module. This is to avoid needing to reason about the equivalence of arbitrary expressions when determining whether two signatures are compatible.

Notice that regexes are spliced in using `$(`e`)`, where e is any expression of type `Regex.t`, while strings are spliced in using `$$(`e`)`, where e is any expression of type **`string`**. These choices are made entire

Every TLM definition also includes a *type annotation*, here **`at`** `html`, and a *parse function* between **`by static`** `{` and `}`.

Using this TLM, we can express the Ur/Web example from Figure 1 as shown in Figure 5c. On both Lines 1 and 2, we apply `$html` to a *generalized literal form* delimited by `[|` and `|]`. Generalized literal forms, which first arose in the prior work on TSLs [53], syntactically subsume other literal forms because the context-free syntax of the language only specifies the outer delimiters. In this paper, we will use `[|` and `|]`, but Omar et al. [53] formally specified several other choices, including layout-sensitive delimitation. *Literal bodies* are constrained only in that `[|` and `|]` must be balanced.

The system delegates responsibility over the parsing and expansion of each literal body to the applied TLM's parse function during a semantic phase called *expansion*, which starts just before and continues into the typing phase.

Because the parse function is applied during this phase, rather than at run-time, we call it a *static function*. Static functions cannot refer to the surrounding variable bindings because those variables stand for run-time values. Instead, there is are separate *static bindings* marked by the **`static`** keyword that populate a *static environment* that is discarded after expansion finishes. A more detailed account of static evaluation, both informal and informal, is given in the supplement. An alternative design that allows for the explicit lowering of standard-phase modules to the static phase has also been proposed for OCaml [72].

The input type of the parse function, `body`, classifies encodings of literal bodies. Literal bodies are sequences of characters, so we define `body` as a synonym of **`string`** in Figure 4. The return type is a sum type, defined by applying the parameterized type `parse_result` defined in Figure 4, that distinguishes between parse errors and successful parses. Let us consider these two possibilities in turn.

If the parse function determines that the literal body is not well-formed according to the syntax that it implements, it must return `ParseError {msg=`e_{msg}`, loc=`e_{loc}`}` where e_{msg} is a custom error message and e_{loc} is a value of type `segment`, defined in Figure 4, that designates a segment of the literal body as the origin of the error [68].

If instead parsing succeeds, the parse function returns `Success` e_{proto}, where e_{proto} is called the *encoding of the proto-expansion*. For expression TLMs, the proto-expansion is a *proto-expression* and it is encoded as a value of the recursive datatype `proto_expr` that is outlined in Figure 4. Most of the constructors of `proto_expr` are individually uninteresting – they encode OCaml's various expression forms. Expressions can mention types, so we also need the type `proto_typ` also outlined in Figure 4. It is only the `SplicedE` and `SplicedT` constructors that are novel. These are discussed next.

2.3 Splicing

When the parse function determines that some segment of the literal body is a spliced expression, according to whatever syntactic criteria it deems suitable, it can indirectly refer to it in the encoding

```
type body = string;
type segment = {startIdx: int, endIdx: int};
type parse_result('a)
  = ParseError {msg: string, loc: segment}
  | Success('a);
type proto_typ = Arrow(proto_typ, proto_typ)
               | StringTy
               | /* ... */
               | SplicedT(segment);
type proto_expr = Tuple(list(proto_expr))
                | /* ... */
                | SplicedE(segment, proto_typ);
```

Fig. 4. Definitions of various types available ambiently to TLM definitions.

it produces using the `SplicedE` constructor of `proto_expr`, which takes a value of type `segment` that indicates the zero-indexed location of the spliced expression relative to the start of the provided literal body. The `SplicedE` constructor also requires a value of type `proto_typ`, which indicates the type that the spliced expression is expected to have. Types can be spliced out by using the `SplicedT` constructor of `proto_typ` analagously.

For example, consider again the two TLM applications in Figure 5c. In each case, the parse function of the `$html` TLM (Figure 5b, Lines 2-4) first sends the literal body through an off-the-shelf HTML parser, `parse_html`. It then passes the result to a function `html_to_ast`, not shown, which produces the corresponding expression encoding of type `proto_expr`. When this function encounters an HTML text node containing matched `{[` and `]}`, then that segment is inserted as a spliced expression of type **`string`**, and similarly text nodes containing matched curly braces produce a spliced expression of type `html`. For instance, the proto-expansion generated for first TLM application in 5c, pretty-printed, is:

```
H1Element(Nil, Cons(TextNode "Hello, ", Cons(
  TextNode spliced<13; 22; string>, Nil)))
```

Here, **`spliced`**`<13; 22;` **`string`**`>` is a reference to the spliced string expression `first_name` by its location relative to the start of the literal body being expanded (the off-the-shelf HTML parser provides the necessary baseline location information for use by `html_to_ast`). It corresponds to the encoding `SplicedE({startIdx=13, endIdx=22}, StringTy)`. Requiring that TLMs refer to spliced expressions indirectly in this manner ensures that a TLM cannot "forge" spliced terms, i.e. claim that some sub-term of the expansion should be given the privileges of a spliced term, discussed in Sec. 2.4, when it does not in fact appear in the literal body.

The *segmentation* inferred from a proto-expansion is the finite set of references to spliced terms contained within. For example, the segmentation inferred from the proto-expression above contains only **`spliced`**`<13; 22;` **`string`**`>`. The system checks that all of the locations in the segmentation are 1) in bounds relative to the literal body; and 2) non-overlapping. This resolves the problem of **Segmentation** described in Sec. 1, i.e. every literal body in a well-formed program has a well-defined segmentation. The TSL mechanism did not maintain this reasoning principle [53]. A program editor or pretty-printer can communicate this segmentation information to the programmer, e.g. by coloring non-spliced segments green as is our convention in this document. In general, spliced expressions might themselves apply TLMs, in which case the convention is to use a distinct color for unspliced segments at each depth. For example, consider a TLM `$smiles` for chemical structures [4] with support for splicing using curly braces:

```
$html [|Chemical structure of sucrose: {
```

```
    $smiles [|{m_glucose}-O-{m_fructose}|]
    |> SMILES.to_svg }|]
```

A program editor or pretty-printer can communicate the type of each spliced expression, also specified abstractly by the segmentation, upon request (for Reason, via Merlin [2].)

2.4 Proto-Expansion Validation

Three important concerns described in Sec. 1 remain: those related to reasoning abstractly about the *hygiene properties*, i.e. **Capture** and **Context Dependence**, and **Typing**. Addressing these concerns is the purpose of the *proto-expansion validation* process, which occurs during the typing phase. Proto-expansion validation results in the *final expansion*, which is simply the proto-expansion with the references to spliced segments replaced with their own final expansions.

2.4.1 Capture. Proto-expansion validation ensures that spliced terms have access *only* to the bindings at the application site—spliced terms cannot capture bindings internal to the proto-expansion. For example, consider the following application site:

```
let tmp = /* ... application site temp ... */;
$html [|<h1>{f(tmp)}</h1>|];
```

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

```
let tmp = /* ... expansion-internal temp ... */;
H1Element(tmp, spliced<5; 10; html>);
```

Naïvely, the binding of the variable `tmp` in the proto-expansion could shadow the application-site binding of `tmp` in the final expansion. To address this problem, splicing is guaranteed to be capture-avoiding. When generating the final expansion, the system discharges the requirement that capture not occur by implicitly alpha-varying the bindings in the proto-expansion as needed. There is no need for TLM providers to explicitly deploy a mechanism that generates fresh variables (as in, e.g., prior *reader macro* systems [20].

Capture avoidance does prevent library providers from intentionally introducing bindings into spliced terms. For example, Haskell's literal notation for monadic values, i.e. **do**-notation, cannot be expressed because it introduces a new binding syntax [33].

In our initial designs, we were able to express this example because we distinguished spliced identifiers, much as we do spliced types and expressions. These could be explicitly bound inside spliced expressions by extending the `SplicedE` and `SplicedT` constructors to take finite sets of spliced identifiers. This is conceptually similar to the approach taken by Herman and Wand [30] for giving (already parsed) sub-terms access to internal variables. However, we ultimately decided to remove this feature because it would disproportionately increase the reasoning burden on clients when they encounter an unfamiliar literal: *might this obscure literal also be introducing bindings?* It is difficult for a program editor to display a set of "hidden bindings" to the programmer. Haskell-style infix notation for monadic values, where bind is `e >>= f`, can be expressed using TLMs without support for spliced identifiers. Reason 3.0 is adopting more concise function syntax, which will decrease the cost of this syntax.

2.4.2 Context Dependence. The proto-expansion validation process also ensures that variables that appear in the proto-expansion do not refer to bindings that appear either at the TLM definition or the application site. In other words, expansions must be completely *context independent* – they can make no assumptions about the surrounding context whatsoever.

A minimal example of a "broken" TLM that does not generate context-independent proto-expansions is below:

```
syntax $broken at t by static {
```

```
    fun(_) => Success (Var "SSTRxESTR") };
```

The proto-expansion that this TLM generates (for any literal body) refers to a variable `x` that it does not itself bind. If proto-expansion validation permitted such a proto-expansion, it would be well-typed only under those application site typing contexts where `x` is bound. This "hidden assumption" makes reasoning about binding and renaming difficult.

Of course, this prohibition does not extend into the spliced terms in a proto-expansion – spliced terms appear at the application site, so they can justifiably refer to application site bindings. (like `first_name` in Fig. 5c.) Because proto-expansions refer to spliced terms indirectly, enforcing context independence is straightforward – we need only that the proto-expansion itself be closed.

Naïvely, this restriction, also present in the prior work on TSLs [53], is quite restrictive – expansions cannot access any library functions. At best, they can require the client to "pass in" required library functions via splicing at every application. In Sec. 4, we will introduce module parameters and partial parameter application to neatly resolve this problem.

2.4.3 Typing. Finally, proto-expansion validation maintains a reasonable typing discipline by (1) checking that the expansion is of the type specified by the TLM's type annotation; (2) checking that each spliced type is valid; (3) checking that the type annotation on each spliced expression is valid; and (4) checking each spliced expression against the specified type annotation. Context independence implies that ML-style type inference can be performed using only the segmentation (because the remainder of an expansion cannot mention the very variables whose types are being inferred). In the prior work on TSLs, spliced terms did not have type annotations

3 Pattern TLMs

Let us now briefly consider the topic of TLMs that generate *patterns*, rather than expressions. Pattern literals are the dual to expression literals in that expression literals support construction whereas pattern literals support deconstruction [50]. For example, we can pattern match on a value `x : html` by applying a pattern TLM `$html` as follows:

```
switch x {
| $html [|<h1>{cs}</h1>|] => /* cs:list(html) */
| _ -> None
};
```

Any list pattern, including one generated by another TLM application, can appear where `cs` appears in the example pattern above. For longer **switch** expressions, the shorthand **`switch`** `x using $html` applies `$html` to every rule where the outermost pattern is of generalized literal form.

Notice that we can use the same name, `$html`, for this pattern TLM as for the expression TLM defined in the previous section. It does not make sense to apply an expression TLM in pattern position (many expression-level constructs, e.g. lambdas, do not correspond even syntactically to patterns), and *vice versa*, so this is unambiguous.

Pattern TLM definitions differ from expression TLM definitions in two ways: (1) a *sort qualifier*, **`for patterns`**, distinguishes them from expression TLM definitions; (2) the return type of the parse function is `parse_result(proto_pat)`, rather than `parse_result(proto_expr)`. The type `proto_pat`, outlined below, classifies encodings of *proto-patterns*.

```
type proto_pat = /* no variable pattern form! */
               | Wild
               | /* ... */
               | SplicedP(segment, proto_typ);
```

The constructor `SplicedP` operates much like `SplicedE` to allow a proto-pattern to refer indirectly to spliced patterns.

To maintain the abstract binding discipline, variable patterns can appear only within spliced patterns. Enforcing this restriction is straightforward: we simply have not defined a variant of the `proto_pat` type that encodes variable patterns (wildcards are allowed.) This restriction ensures that only variables visible to the client in a spliced pattern are bound in the corresponding branch expression. This is analagous to the capture avoidance principle for expression TLMs.

Type annotations on references to spliced patterns could refer to type variables, so we also need to enforce context independence in the manner discussed in the previous section.

To maintain an abstract typing discipline, proto-pattern validation checks type annotations much as in Sec. 2.4.3.

4 Parametric TLMs

The simple TLMs in the previous sections operate only at one specified type, as did the TSLs in the prior work [53]. This is rather limiting. This section introduces *parametric TLMs*, which can operate over a type- and module-parameterized family of types. They also neatly solve the problem discussed in Sec. 2.4.2 of giving expansions access to helper functions.

Consider the following Reason/OCaml module type (a.k.a. signature), which specifies an abstract data type [25, 38] of string-keyed polymorphic dictionaries:

```
module type DICT = {
  type t('a);
  let empty : t('a);
  let extend : t('a) -> (string, 'a) -> t('a);
  /* ... */
};
```

We can define a TLM that is parametric over implementations of this signature, `D : DICT`, and over choices of the codomain type, `'a` as follows:

```
syntax $dict (D : DICT) (type 'a) at D.t('a)
by static { fun(b) => /* ... */ };
```

For example, given some module that implements this signature, `HashDict : DICT`, we can apply `$dict` as follows:

```
$dict HashDict int [|"key1"=>10; "key2"=>15|]
```

Notice that the segmentation immediately reveals which punctuation is particular to this TLM and where the spliced key and value expressions appear. Because the context-free syntax of unexpanded terms is never modified, it is possible to reason modularly about syntactic determinism, i.e. we can reason above that => does not appear in the follow set of unexpanded expressions [60], so there can never be an ambiguity about where a key expression ends.

The proto-expansion generated for the TLM application above, shown below, can refer to the parameters:

```
D.extend(D.extend D.empty (spliced<1;6;string>,
  spliced<11;12;'a>)) (spliced<15;20;string>,
  spliced<25;26;'a>)
```

Validation checks that the proto-expansion is truly parametric, i.e. it must be valid for all modules `D : DICT` and types `'a`. It is only after validation that we substitute the actual parameters, here `HashDict` for `D` and `int` for `'a`, into the final expansion. Only the type annotations on references to spliced terms are subject to early parameter substitution (because they classify application site terms.)

If we will use the `HashDict` implementation ubiquitously, we can abbreviate the partial application of `$dict` to `HashDict`, resulting in a TLM that is parametric over only the type `'a`:

```
type html_attrs = list(string, string)
type html = BodyElement(html_attrs, list(html))
          | H1Element(html_attrs, list(html))
          | TextNode(string)
          | /* ... */;
```

(a) The html type, which classifies encodings of HTML data.

```
syntax $html at html by static {
  fun(b : body) : parse_result(proto_expr) =>
    try (parse_html b |> html_to_ast |> Success
    | InvalidHTML msg loc => ParseError msg loc
};
```

(b) The $html TLM definition. Figure 4 defines TLM-related types.

```
let heading first_name = $html [|<h1>Hello, {[first_name]}!</h1>|]
let body = $html [|<body>{heading "World!"} ...</body>|]
```

(c) Two examples of the $html TLM being applied. Compare to Ur/Web's built in HTML literal notation from Figure 1.

Fig. 5. Case Study: HTML literals

```
syntax $dict' = $dict HashDict;
```

TLM abbreviations can themselves be parameterized to support partial application of parameters other than the last.

In Sec. 2.4.2 we discussed the problem of the strict context independence discipline being too restrictive, in that it would seem to restrict expansions from referring to useful helpers bound at the TLM definition site. Module parameters address this problem – the helper values, types and modules can be packaged into a module, passed in and partially applied to hide this detail from clients. Because this will be common in practice, we provide the following shorthand:

```
syntax $a at t using X~${}_1$~=M~${}_1$~,...,X~${}_n$~=M~${}_n$~ by ...
```

This explicit parameter passing discipline is reminiscent of work on explicit capture for distributed functions [47]. By not implicitly giving expansions access to all definition-site bindings, we need not examine parse functions to reason about, e.g., renaming. Consequently, encodings of proto-terms can be values of standard datatypes (e.g. proto_expr) with variables represented simply as, e.g., strings, and quasiquotation notation can be expressing using TLMs (see supplement). A central design goal of Reason is to leave the OCaml semantics unchanged, so building quasiquotation in primitively, to integrate the free variables in term encodings into the overall binding discipline [5], was in any case infeasible.

5 Additional Examples

5.1 HTML

We start in this section by describing the TLM mechanism in detail by way of a substantial case study: literal notation, like that built in to Ur/Web (see Figure 1) and also, presently, Reason, for expressions of the type html in Figure 5a.

5.2 Quotation

5.3 More Examples, Briefly

6 Simple TLMs, Formally

This section will present a calculus of simple expression and pattern TLMs called $\lambda^{\mathbf{S}}_{\mathbf{TLM}}$. By the end of this section, we will have a theorem that encodes the five reasoning principles that were outlined informally in the previous sections.

Because our focus is on these reasoning principles, and for reasons of space, we leave our full calculus of parametric TLMs to the supplement. The full calculus extends the simple calculus of this section with an ML module calculus based closely on the system defined by Harper [26], which

$$
\begin{array}{llcl}
\text{UTyp} & \hat{\tau} & ::= & \hat{t} \mid \hat{\tau} \rightharpoonup \hat{\tau} \mid \forall \hat{t}.\hat{\tau} \mid \mu \hat{t}.\hat{\tau} \\
 & & \mid & \langle \{i \hookrightarrow \hat{\tau}_i\}_{i \in L} \rangle \mid [\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}] \\
\text{UExp} & \hat{e} & ::= & \hat{x} \mid \lambda \hat{x}{:}\hat{\tau}.\hat{e} \mid \hat{e}(\hat{e}) \mid \Lambda \hat{t}.\hat{e} \mid \hat{e}[\hat{\tau}] \mid \texttt{fold}(\hat{e}) \\
 & & \mid & \langle \{i \hookrightarrow \hat{e}_i\}_{i \in L} \rangle \mid \texttt{inj}[\ell](\hat{e}) \mid \texttt{match}\ \hat{e}\ \{\hat{r}_i\}_{1 \le i \le n} \\
 & & \mid & \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for expressions by static}\ e\ \texttt{in}\ \hat{e} \\
 & & \mid & \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for patterns by static}\ e\ \texttt{in}\ \hat{e} \\
 & & \mid & \hat{a}\ [\![b]\!] \\
\text{URule} & \hat{r} & ::= & \hat{p} \Rightarrow \hat{e} \\
\text{UPat} & \hat{p} & ::= & \hat{x} \mid \_ \mid \texttt{fold}(\hat{p}) \mid \langle \{i \hookrightarrow \hat{p}_i\}_{i \in L} \rangle \mid \texttt{inj}[\ell](\hat{p}) \mid \hat{a}\ [\![b]\!]
\end{array}
$$

Fig. 6. Syntax of the $\lambda^{\mathbf{S}}_{\mathbf{TLM}}$ unexpanded language (UL). Metavariable $\hat{t}$ ranges over type identifiers, $\hat{x}$ over expression identifiers, ℓ over labels, L over finite sets of labels, $\hat{a}$ over TLM names and b over literal bodies. We write $\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}$ for a finite mapping of each label i in L to some unexpanded type $\hat{\tau}_i$, and similarly for other sorts. We write $\{\hat{r}_i\}_{1 \le i \le n}$ for a finite sequence of n unexpanded rules.

in turn is based on early work by MacQueen [41, 42], subsequent work on the phase splitting interpretation of modules [27] and on using dependent singleton kinds to track type identity [12, 64], and finally on formal developments by Dreyer [15] and Lee et al. [36]. These additional mechanisms are necessary only to formalize the advanced features of Sec. 4. Proofs are in the supplement for both the simple and full calculus.

Both the simple and full calculus consist of an *unexpanded language*, or *UL*, defined by typed expansion to an *expanded language*, or *XL*. Figs. 6 and 7 summarize the syntax of the UL and the XL, respectively.

6.1 Expanded Language (XL)

The XL of $\lambda^{\mathbf{S}}_{\mathbf{TLM}}$ forms a standard pure functional language with partial function types, quantification over types, recursive types, labeled product types and labeled sum types and support for pattern matching. The reader is directed to *PFPL* [26] for a detailed introductory account of these constructs. We will only tersely summarize the statics and dynamics of the XL below because the particularities are not critical.

The *statics of the XL* 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, $\Delta \vdash p : \tau \dashv \Gamma$. In the latter, Γ is a typing context that tracks the typing hypotheses generated by p. These judgements are inductively defined in the supplemental material along with necessary auxiliary structures and standard lemmas.

The *evaluation semantics* of $\lambda^{\mathbf{S}}_{\mathbf{TLM}}$ 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'.

6.2 Syntax of the Unexpanded Language

Unexpanded types and expressions are simple inductive structures. Unlike expanded types and expressions, they are **not** abstract binding trees – we do **not** define the standard notions of renaming, alpha-equivalence or substitution for unexpanded terms. This is because unexpanded expressions remain "partially parsed" due to the presence of literal bodies, b, from which spliced terms might be extracted during typed expansion. In fact, unexpanded types and expressions do not involve variables at all, but rather *type identifiers*, $\hat{t}$, and *expression identifiers*, $\hat{x}$. Identifiers are given meaning by expansion to variables during typed expansion, as we will see. This distinction between identifiers and variables is technically crucial to our developments.

Most of the unexpanded forms in Figure 6 mirror the expanded forms. We refer to these as the *common forms*.

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

Fig. 7. Syntax of the expanded language (XL). XL terms are *abstract binding trees* (ABTs) identified up to alpha-equivalence, so we follow the syntactic conventions of Harper [26]. Metavariable x and t ranges over variables.

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

Fig. 8. An example expansion derviation demonstrating how identifiers and variables are separately tracked.

There is also a corresponding context-free textual syntax for the UL. Giving a complete definition of the context-free textual syntax as, e.g., a context-free grammar, is not critical to our purposes here. 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 (the supplement states the full condition.)

6.3 Typed Expansion

Unexpanded terms are checked and expanded simultaneously according to the central *typed expansion judgements*:

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

The typed expansion rules that handle common forms mirror the corresponding typing rules. The *expression TLM context*, $\hat{\Psi}$, and the *pattern TLM context*, $\hat{\Phi}$, pass through these rules opaquely. For example, the rules for variables and lambdas are shown being applied in Fig. 8, discussed below.

The only subtlety related to common forms has to do with the relationship between identifiers, $\hat{x}$, in the UL and variables, x, in the XL. To understand this, we must first describe in detail how unexpanded contexts work. *Unexpanded typing contexts*, $\hat{\Gamma}$, are pairs of the form $\langle \mathcal{G};\Gamma\rangle$, where $\mathcal{G}$ maps each expression identifier $\hat{x} \in \mathrm{dom}(\mathcal{G})$ to the hypothesis $\hat{x} \leadsto x$, for some expression variable, x, called its expansion. The standard typing context, Γ, then tracks the type of x. We write $\mathcal{G} \uplus \hat{x} \leadsto x$ for the expression identifier expansion context that maps $\hat{x}$ to $\hat{x} \leadsto x$ and defers to $\mathcal{G}$ for all other expression identifiers (i.e. the previous mapping is **updated**.) Note the distinction between update and extension (which requires that the new identifier is not already in the domain.) We define $\hat{\Gamma}, \hat{x} \leadsto x : \tau$ when $\hat{\Gamma} = \langle \mathcal{G};\Gamma\rangle$ as an abbreviation of $\langle \mathcal{G} \uplus \hat{x} \leadsto x; \Gamma, x : \tau\rangle$.

To develop an intuition for why the update operation is necessary, it is instructive to inspect in Fig. 8 the derivation of the expansion of the unexpanded expression $\lambda\hat{x}{:}\hat{\tau}.\lambda\hat{x}{:}\hat{\tau}.\hat{x}$ to $\mathtt{lam}\{\tau\}(x_1.\mathtt{lam}\{\tau\}(x_2.x_2))$ assuming $\hat{\Delta} \vdash \hat{\tau} \leadsto \tau$ type. Notice that when Rule ee-lam is applied, the type identifier expansion context is updated but the typing context is extended with a

$$\textsc{ee-def-setsm}\quad \frac{\hat{\Delta} \vdash \hat{\tau} \leadsto \tau\ \mathsf{type} \qquad \emptyset\ \emptyset \vdash e_{\text{parse}} : \mathtt{parr(Body;ParseResultSE)} \qquad e_{\text{parse}} \Downarrow e'_{\text{parse}} \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}, \hat{a} \leadsto a \hookrightarrow \mathtt{setlm}(\tau;\, e'_{\text{parse}}); \hat{\Phi}} \hat{e} \leadsto e : \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{for\ expressions\ by\ static}\ e_{\text{parse}}\ \mathtt{in}\ \hat{e} \leadsto e : \tau'}$$

$$\textsc{ee-ap-setsm}\quad \frac{b \downarrow_{\text{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \mathtt{inj[Succ} \qquad e_{\text{proto}} \uparrow_{\text{PrExpr}} \grave{e} \qquad \mathsf{seg}(\grave{e})\ \mathsf{segme} \qquad \emptyset\ \emptyset \vdash^{\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; \hat{\Phi}; b} \grave{e} \leadsto e : \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}', \hat{a} \leadsto a \hookrightarrow \mathtt{setlm}(\tau;\, e_{\text{parse}}); \hat{\Phi}} \hat{a}\ [\![b]}$$

Fig. 9. The typed expansion rules for expression TLM definition and application.

(necessarily fresh) variable, first x_1 then x_2. Without this mechanism, expansions for unexpanded terms with shadowing, like this minimal example, would not exist, because we cannot implicitly alpha-vary the unexpanded term to sidestep this problem in the usual manner.

6.4 TLM Definitions

Rule ee-def-setsm in Fig. 9 governs simple expression TLM (seTLM) definitions. The first premise expands the unexpanded type annotation. The second premise checks that e_{parse} is a closed expanded function of the given function type. (In the supplement, we add the machinery necessary for parse functions that are neither closed nor yet expanded.)

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 $b \downarrow_{\text{Body}} e_{\text{body}}$ that will come up below.

The return type, `ParseResultSE`, abbreviates a labeled sum type that distinguishes parse errors from successful parses: $\mathtt{ParseError} \hookrightarrow \langle\rangle, \mathtt{SuccessE} \hookrightarrow \mathtt{PrExpr}$.

The type abbreviated `PrExpr` classifies encodings of *proto-expressions*, $\grave{e}$ (pronounced "grave e".) The syntax of proto-expressions, defined in Figure 10, will be described when we describe proto-expansion validation in Sec. 6.6. The mapping from proto-expressions to values of type `PrExpr` is defined by the *proto-expression encoding judgement*, $\grave{e} \downarrow_{\text{PrExpr}} e$. An inverse mapping is defined by the *proto-expression decoding judgement*, $e \uparrow_{\text{PrExpr}} \grave{e}$. Again, we need only take as a condition that there is an isomorphism between values of type `PrExpr` and closed proto-expressions mediated by these judgements (see supplement.)

The third premise of Rule ee-def-setsm evaluates the parse function to a value. This is not necessary, but it is the choice one would expect to make in an eager language.

The final premise of Rule ee-def-setsm extends the expression TLM context, $\hat{\Psi}$, with the newly determined seTLM definition, and proceeds to assign a type, τ', and expansion, e, to $\hat{e}$. The conclusion of the rule then assigns this type and expansion to the seTLM definition as a whole.

Expression TLM contexts, $\hat{\Psi}$, are of the form $\langle \mathcal{A}; \Psi \rangle$, where $\mathcal{A}$ is a *TLM identifier expansion context* and Ψ is an *expression TLM definition context*. We distinguish TLM identifiers, $\hat{a}$, from TLM names, a, for much the same reason that we distinguish type and expression identifiers from type and expression variables: in order to allow a TLM definition to shadow a previously defined TLM definition without relying on an implicit identification convention.

An expression TLM definition context, Ψ, is a finite function mapping each TLM name $a \in \text{dom}(\Psi)$ to an *expanded seTLM definition*, $a \hookrightarrow \mathtt{setlm}(\tau;\, e_{\text{parse}})$, where τ is the seTLM's type annotation, and e_{parse} is its parse function. We define $\hat{\Psi}, \hat{a} \leadsto a \hookrightarrow \mathtt{setlm}(\tau;\, e_{\text{parse}})$, when $\hat{\Psi} = \langle \mathcal{A}; \Psi \rangle$, as an abbreviation of $\langle \mathcal{A} \uplus \hat{a} \leadsto a; \Psi, a \hookrightarrow \mathtt{setlm}(\tau;\, e_{\text{parse}}) \rangle$.

The simple pattern TLM (spTLM) definition form operates analagously (see supplement), with the spTLM context, $\hat{\Phi}$, rather than the $\hat{\Psi}$ updated. This allows expression and pattern TLMs to use the same identifiers.

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

Fig. 10. Syntax of proto-expansions. Proto-expansion terms are ABTs identified up to alpha-equivalence.

6.5 TLM Application

The unexpanded expression form for applying an seTLM named $\hat{a}$ to a literal form with literal body b is $\hat{a}$ $[\![b]\!]$. Rule EE-AP-SETSM governing this form is shown in Fig. 9.

The first premise encodes the literal body, e_{body}, which, as described above, is a value of type `Body`.

The second 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` (assuming a well-formed expression TLM context, by application of Type Safety.) We call e_{proto} the *encoding of the proto-expansion*. If the parse function produces a value labeled `ParseError`, then typed expansion fails and formally, no rule is necessary.

The third premise decodes the encoding of the proto-expansion using the judgement described in Sec. 6.4.

The fourth premise of Rule EE-AP-SETSM determines the segmentation of the proto-expansion, seg($\grave{e}$), and ensures that it is valid with respect to b via the predicate ψ segments b, which checks that each segment in the finite set of segments ψ has non-negative length and is within bounds of b, and that the segments in ψ do not overlap.

The final premise *validates* the proto-expansion and simultaneously generates the *final expansion*, e, which appears in the conclusion of the rule. The proto-expression validation judgement is defined in the next subsection.

The typed pattern expansion rule governing pattern TLM application is analagous (see supplement).

6.6 Proto-Expansion Validation

Finally, we arrive at the crucial *proto-expansion validation judgements*, which validate the proto-expansions generated by TLMs and simultaneously generate their final expansions:

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

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-SETSM in Fig. 9) 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. 10 mirror corresponding expanded forms. The rules governing proto-expansion validation for these common forms in the supplement 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. This maintains context independence (defined formally below.)

Notice that proto-rules, $\grave{r}$, involve expanded patterns, p, not proto-patterns, $\grave{p}$. The reason is that proto-rules appear in proto-expressions, which are generated by expression TLMs. Proto-patterns, in contrast, arise only from pattern TLMs. There is not a variable proto-pattern form, for the reasons described in Sec. 3.

References to Spliced Terms The only interesting forms are the references to spliced unexpanded types, expressions and patterns. Let us consider the rule for references to spliced unexpanded expressions:

PEV-SPLICED

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

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 third premise then invokes type expansion on $\hat{e}$ *under the application site contexts*, $\langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle$ and $\langle \mathcal{G}; \Gamma_{\mathsf{app}} \rangle$, but *not* the expansion-local 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.

The rule for references to spliced unexpanded types and patterns are fundamentally analagous (see supplement).

6.7 Metatheory

6.7.1 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 6.1 (TYPED EXPRESSION EXPANSION (STRONG)).

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

(2) If $\Delta\ \Gamma \vdash^{\langle \mathcal{D};\Delta_{app} \rangle;\, \langle \mathcal{G};\Gamma_{app} \rangle;\, \langle \mathcal{A};\Psi \rangle;\, b} \grave{e} \leadsto e : \tau$ *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, not in the final expansion.) The combined context can only be formed if these are disjoint.

The proof proceeds by mutual rule induction and appeal to simple lemmas about type expansion and proto-type validation (see supplement). The proof is straightforward but for one issue: 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 supplemental material.

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

THEOREM 6.2 (SETLM REASONING PRINCIPLES).

If $\langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}} \hat{a}\ [\![b]\!] \leadsto e : \tau$ *then:*

(1) ***(Typing 1)*** $\hat{\Psi} = \hat{\Psi}', \hat{a} \leadsto a \hookrightarrow \mathtt{setlm}(\tau; e_{parse})$ *and* $\Delta\ \Gamma \vdash e : \tau$

The type of the expansion is consistent with the type annotation on the applied seTLM definition.

(2) $b \downarrow_{\mathsf{Body}} e_{body}$

(3) ***(Responsibility)*** $e_{parse}(e_{body}) \Downarrow \texttt{inj[SuccessE]}(e_{proto})$
The parse function of the invoked TLM is responsible for the expansion.

(4) $e_{proto} \uparrow_{\mathsf{PrExpr}} \grave{e}$

(5) ***(Segmentation)*** $\mathsf{seg}(\grave{e})$ segments b
The segmentation determined by the proto-expansion actually segments the literal body (i.e. each segment is in-bounds and the segments are non-overlapping.)

(6) $\mathsf{seg}(\grave{e}) = \{\texttt{splicedt}[m_i'; n_i']\}_{0 \leq i < n_{ty}} \cup \{\texttt{splicede}[m_i; n_i; \grave{\tau}_i]\}_{0 \leq i < n_{exp}}$

(7) ***(Typing 2)*** $\{\langle \mathcal{D}; \Delta \rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m_i'; n_i')) \rightsquigarrow \tau_i' \;\mathsf{type}\}_{0 \leq i < n_{ty}}$ *and* $\{\Delta \vdash \tau_i' \;\mathsf{type}\}_{0 \leq i < n_{ty}}$
Each spliced type has a well-formed expansion.

(8) ***(Typing 3)*** $\{\emptyset \vdash^{\langle \mathcal{D}; \Delta \rangle; b} \grave{\tau}_i \rightsquigarrow \tau_i \;\mathsf{type}\}_{0 \leq i < n_{exp}}$ *and* $\{\Delta \vdash \tau_i \;\mathsf{type}\}_{0 \leq i < n_{exp}}$
Each type annotation on a reference to a spliced expression has a well-formed expansion.

(9) ***(Typing 4)***
$\{\langle \mathcal{D}; \Delta \rangle \; \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}} \mathsf{parseUExp}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow e_i : \tau_i\}_{0 \leq i < n_{exp}}$ *and* $\{\Delta \; \Gamma \vdash e_i : \tau_i\}_{0 \leq i < n_{exp}}$
Each spliced expression has a well-typed expansion consistent with the type annotation in the segmentation.

(10) ***(Capture Avoidance)***
$e = [\{\tau_i'/t_i\}_{0 \leq i < n_{ty}}, \{e_i/x_i\}_{0 \leq i < n_{exp}}]e'$ *for some variables* $\{t_i\}_{0 \leq i < n_{ty}}$ *and* $\{x_i\}_{0 \leq i < n_{exp}}$, *and* e'
The final expansion can be decomposed into a term with variables in place of each spliced type or expression. The expansions of these spliced types and expressions can be substituted into this term in the standard capture avoiding manner.

(11) ***(Context Independence)***
$\mathsf{fv}(e') \subset \{t_i\}_{0 \leq i < n_{ty}} \cup \{x_i\}_{0 \leq i < n_{exp}}$
The decomposed term makes no mention of bindings in the application site context, i.e. the only free variables 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 [3, 11, 16, 29, 30, 35]. 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, rather than going straight to the final expansion.

The reasoning principles theorem for pattern TLMs is in the supplement. The key clause establishes that the hypotheses generated by the TLM application form are exactly the union of the hypothesis generated by the spliced patterns.

In the full calculus, which supports parametric TLMs, the context independence clause allows reference to the variables standing for parameters.

7 Parametric TLMs, Formally

We will now outline $\lambda^{\mathbf{P}}_{\mathbf{TLM}}$, a calculus that extends $\lambda^{\mathbf{S}}_{\mathbf{TLM}}$ with parametric TLMs. This calculus is organized, like $\lambda^{\mathbf{S}}_{\mathbf{TLM}}$, as an unexpanded language (UL) defined by typed expansion to an expanded language (XL). There is not enough space to describe $\lambda^{\mathbf{P}}_{\mathbf{TLM}}$ with the same level of detail as in Sec. 6, so we highlight only the most important concepts below. The details are in the supplement.

The XL consists of 1) module expressions, M, classified by signatures, σ; 2) constructions, c, classified by kinds, κ; and 3) expressions classified by types, which are constructions of kind Type (we use metavariables τ instead of c for types by convention.) Metavariables X ranges over module variables and u or t over construction variables. The module and construction languages are based closely on those defined by Harper [26], which in turn are based on early work by MacQueen [41, 42], subsequent work on the phase splitting interpretation of modules [27] and on using dependent singleton kinds to track type identity [12, 64], and finally on formal developments by Dreyer [15] and Lee et al. [36]. A complete account of these developments is unfortunately beyond the scope of this paper. The expression language extends the language of $\lambda^{\mathbf{S}}_{\mathbf{TLM}}$ only to allow projection out of modules.

The main conceptual difference between $\lambda^{\mathbf{S}}_{\mathbf{TLM}}$ and $\lambda^{\mathbf{P}}_{\mathbf{TLM}}$ is that $\lambda^{\mathbf{P}}_{\mathbf{TLM}}$ introduces the notion of unexpanded and expanded TLM expressions and types, as shown in Fig. 11.

$$\begin{array}{llcl@{\qquad}llcl}
\mathsf{UMType} & \hat{\rho} & ::= & \hat{\tau} \mid \forall \hat{X}{:}\hat{\sigma}.\hat{\rho} & \mathsf{MType} & \rho & ::= & \mathtt{type}(\tau) \mid \mathtt{allmods}\{\sigma\}(X.\rho) \\
\mathsf{UMExp} & \hat{\epsilon} & ::= & \hat{a} \mid \Lambda \hat{X}{:}\hat{\sigma}.\hat{\epsilon} \mid \hat{\epsilon}(\hat{X}) & \mathsf{MExp} & \epsilon & ::= & \mathtt{defref}[a] \mid \mathtt{absmod}\{\sigma\}(X.\epsilon) \mid \mathtt{apmod}\{X\}(\epsilon)
\end{array}$$

Fig. 11. Syntax of unexpanded and expanded TLM types and expressions in $\lambda^{\mathbf{P}}_{\mathbf{TLM}}$

The TLM type $\mathtt{allmods}\{\sigma\}(X.\rho)$ classifies TLM expressions that have one module parameter matching σ. For simplicity, we formalize only module parameters. Type parameters can be expressed as module parameters having exactly one abstract type member.

The rule governing expression TLM application, reproduced below, touches all of the main ideas in $\lambda^{\mathbf{P}}_{\mathbf{TLM}}$, so we will refer to it throughout the remainder of this section.

EE-AP-PETSM

$$\frac{\begin{array}{c}
\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\text{app}} \rangle \qquad \hat{\Psi} = \langle \mathcal{A}; \Psi \rangle \\
\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \mathtt{type}(\tau_{\text{final}}) \qquad \Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \Downarrow \epsilon_{\text{normal}} \\
\mathsf{tlmdef}(\epsilon_{\text{normal}}) = a \qquad \Psi = \Psi', a \hookrightarrow \mathtt{petlm}(\rho; e_{\text{parse}}) \\
b \downarrow_{\mathsf{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \mathtt{inj[SuccessE]}(e_{\text{pproto}}) \qquad e_{\text{pproto}} \uparrow_{\mathsf{PPrExpr}} \dot{e} \\
\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \dot{e} \looparrowright_{\epsilon_{\text{normal}}} \grave{e} \mathrel{?} \mathtt{type}(\tau_{\text{proto}}) \dashv \omega : \Omega_{\text{params}} \\
\mathsf{seg}(\grave{e}) \mathsf{~segments~} b \qquad \Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}};\, \hat{\Omega};\, \hat{\Psi};\, \hat{\Phi};\, b} \grave{e} \rightsquigarrow e : \tau_{\text{proto}}
\end{array}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{\epsilon} \llbracket b \rrbracket \rightsquigarrow [\omega]e : [\omega]\tau_{\text{proto}}}$$

The first two premises simply deconstruct the (unified) unexpanded context $\hat{\Omega}$ (which tracks the expansion of expression, constructor and module identifiers, as $\hat{\Delta}$ and $\hat{\Gamma}$ did in $\lambda^{\mathbf{S}}_{\mathbf{TLM}}$) and peTLM context, $\hat{\Psi}$. Next, we expand $\hat{\epsilon}$ according to straightforward unexpanded peTLM expression expansion rules. The resulting TLM expression, ϵ, must be defined at a type (i.e. no quantification over modules must remain once the literal body is encountered.)

The fourth premise performs *peTLM expression normalization*, $\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \Downarrow \epsilon_{\text{normal}}$. This is defined in terms of a structural operational semantics [57] with two stepping rules:

EPS-DYN-APMOD-SUBST-E

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{apmod}\{X\}(\mathtt{absmod}\{\sigma\}(X'.\epsilon)) \mapsto [X/X']\epsilon}$$

EPS-DYN-APMOD-STEPS-E

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto \epsilon'}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{apmod}\{X\}(\epsilon) \mapsto \mathtt{apmod}\{X\}(\epsilon')}$$

Normalization eliminates parameters introduced in higher-order abbreviations, leaving only those parameter applications specified by the original TLM definition. Normal forms and progress and preservation theorems are established in the supplement.

The third row of premises looks up the applied TLM's definition by invoking a simple metafunction to extract its name, a, then looking up a within the peTLM definition context, Ψ.

The fourth row of premises 1) encodes the body as a value of the type Body; 2) applies the parse function; and 3) decodes the result, producing a *parameterized proto-expression*, $\dot{e}$. Parameterized proto-expressions, $\dot{e}$, are ABTs that serve simply to introduce the parameter bindings into an underlying proto-expression, $\grave{e}$. The syntax of parameterized proto-expressions is given below.

$$\mathsf{PPrExp} \quad \dot{e} \quad ::= \quad \mathtt{prexp}(\grave{e}) \mid \mathtt{prbindmod}(X.\dot{e})$$

There must be one binder in $\dot{e}$ for each TLM parameter specified by a. (In Reason, we can insert these binders automatically as a convenience.)

The judgement on the fifth row of Rule EE-AP-PETSM then *deparameterizes* $\dot{e}$ by peeling away these binders to produce 1) the underlying proto-expression, $\grave{e}$, with the variables that stand for the parameters free; 2) a corresponding deparameterized type, τ_{proto}, that uses the same free variables to stand for the parameters; 3) a *substitution*, ω, that pairs the applied parameters from ϵ_{normal} with the corresponding variables generated when peeling away the binders in $\dot{e}$; and 4) a corresponding *parameter context*, Ω_{params}, that tracks the signatures of these variables. The two rules governing the proto-expression deparameterization judgement are below:

$$\frac{}{\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi, a \hookrightarrow \mathtt{petlm}(\rho; e_{\text{parse}})} \mathtt{prexp}(\grave{e}) \hookleftarrow_{\mathtt{defref}[a]} \grave{e} \mathrel{?} \rho \dashv \emptyset : \emptyset}$$

$$\frac{\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \dot{e} \hookleftarrow_{\epsilon} \grave{e} \mathrel{?} \mathtt{allmods}\{\sigma\}(X.\rho) \dashv \omega : \Omega \qquad X \notin \text{dom}(\Omega_{\text{app}})}{\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{prbindmod}(X.\dot{e}) \hookleftarrow_{\mathtt{apmod}\{X'\}(\epsilon)} \grave{e} \mathrel{?} \rho \dashv (\omega, X'/X) : (\Omega, X : \sigma)}$$

This judgement can be pronounced "when applying peTLM ϵ, $\dot{e}$ has deparameterization $\grave{e}$ leaving ρ with parameter substitution ω". Notice based on the second rule that every module binding in $\dot{e}$ must pair with a corresponding module parameter application. Moreover, the variables standing for parameters must not appear in Ω_{app}, i.e. $\text{dom}(\Omega_{\text{params}})$ must be disjoint from $\text{dom}(\Omega_{\text{app}})$ (this requirement can always be discharged by alpha-variation.)

The final row of premises checks that the segmentation of $\grave{e}$ is valid and performs proto-expansion validation under the parameter context, Ω_{param} (rather than the empty context, as was the case in $\lambda^{\mathbf{S}}_{\mathbf{TLM}}$.) The conclusion of the rule applies the parameter substitution, ω, to the resulting expression and the deparameterized type.

Proto-expansion validation operates conceptually as in $\lambda^{\mathbf{S}}_{\mathbf{TLM}}$. The only subtlety has to do with the type annotations on references to spliced terms. As described at the end of Sec. **??**, these annotations might refer to the parameters, so the parameter substitution, ω, which is tracked by the splicing scene, must be applied to the type annotation before proceeding recursively to expand the referenced unexpanded term. However, the spliced term itself must treat parameters parametrically, so the substitution is not applied in the conclusion of the following rule:

$$\frac{\begin{array}{c}\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e} \qquad \Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}};\, \hat{\Omega};\, b} \grave{\tau} \rightsquigarrow \tau :: \mathsf{Type} \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : [\omega]\tau \\ \hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\text{app}} \rangle \qquad \text{dom}(\Omega) \cap \text{dom}(\Omega_{\text{app}}) = \emptyset\end{array}}{\Omega \vdash^{\omega:\Omega_{\text{params}};\, \hat{\Omega};\, \hat{\Psi};\, \hat{\Phi};\, b} \mathtt{splicede}[m; n; \grave{\tau}] \rightsquigarrow e : \tau}$$

(This is only sensible because we maintain the invariant that Ω is always an extension of Ω_{params}.)

The calculus enjoys metatheoretic properties analagous to those described in Sec. 6.7, modified to account for the presence of modules, kinds and parameterization. The following theorem establishes the abstract reasoning principles available when applying a parametric expression TLM. The clauses are directly analagous to those of Theorem 6.2, so for reasons of space we do not repeat the inline descriptions. The **Kinding** clauses can be understood by analogy to the **Typing** clauses. The details of parametric pattern TLMs (ppTLMs) are analagous (see supplement.)

Theorem 7.1 (peTLM Reasoning Principles). *If* $\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}\ [\![b]\!] \rightsquigarrow e : \tau$ *then:*

(1) $\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{app} \rangle$
(2) $\hat{\Psi} = \langle \mathcal{A}; \Psi \rangle$
(3) **(Typing 1)** $\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{\epsilon} \rightsquigarrow \epsilon$ @ $\mathtt{type}(\tau)$ *and* $\Omega_{app} \vdash e : \tau$
(4) $\Omega_{app} \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \Downarrow \epsilon_{normal}$
(5) $\mathsf{tlmdef}(\epsilon_{normal}) = a$
(6) $\Psi = \Psi', a \hookrightarrow \mathtt{petlm}(\rho; e_{parse})$
(7) $b \downarrow_{\mathsf{Body}} e_{body}$
(8) $e_{parse}(e_{body}) \Downarrow \mathtt{inj[SuccessE]}(e_{pproto})$
(9) $e_{pproto} \uparrow_{\mathsf{PPrExpr}} \grave{e}$
(10) $\Omega_{app} \vdash^{\mathsf{Exp}}_{\Psi} \grave{e} \looparrowright_{\epsilon_{normal}} \grave{e}\ ?\ \mathtt{type}(\tau_{proto}) \dashv \omega : \Omega_{params}$
(11) **(Segmentation)** $\mathsf{seg}(\grave{e})$ segments b
(12) $\Omega_{params} \vdash^{\omega:\Omega_{params};\ \hat{\Omega};\ \hat{\Psi};\ \hat{\Phi};\ b} \grave{e} \rightsquigarrow e' : \tau_{proto}$
(13) $e = [\omega]e'$
(14) $\tau = [\omega]\tau_{proto}$
(15) $\mathsf{seg}(\grave{e}) = \{\mathtt{splicedk}[m_i; n_i]\}_{0 \le i < n_{kind}} \cup \{\mathtt{splicedc}[m'_i; n'_i; \grave{\kappa}'_i]\}_{0 \le i < n_{con}} \cup \{\mathtt{splicede}[m''_i; n''_i; \grave{\tau}_i]\}_{0 \le i < n_{exp}}$
(16) **(Kinding 1)** $\{\hat{\Omega} \vdash \mathsf{parseUKind}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow \kappa_i\ \mathsf{kind}\}_{0 \le i < n_{kind}}$ *and* $\{\Omega_{app} \vdash \kappa_i\ \mathsf{kind}\}_{0 \le i < n_{kind}}$
(17) **(Kinding 2)** $\{\Omega_{params} \vdash^{\omega:\Omega_{params};\ \hat{\Omega};\ b} \grave{\kappa}'_i \rightsquigarrow \kappa'_i\ \mathsf{kind}\}_{0 \le i < n_{con}}$ *and* $\{\Omega_{app} \vdash [\omega]\kappa'_i\ \mathsf{kind}\}_{0 \le i < n_{con}}$
(18) **(Kinding 3)** $\{\hat{\Omega} \vdash \mathsf{parseUCon}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow c_i :: [\omega]\kappa'_i\}_{0 \le i < n_{con}}$ *and* $\{\Omega_{app} \vdash c_i :: [\omega]\kappa'_i\}_{0 \le i < n_{con}}$
(19) **(Kinding 4)** $\{\Omega_{params} \vdash^{\omega:\Omega_{params};\ \hat{\Omega};\ b} \grave{\tau}_i \rightsquigarrow \tau_i :: \mathsf{Type}\}_{0 \le i < n_{exp}}$ *and* $\{\Omega_{app} \vdash [\omega]\tau_i :: \mathsf{Type}\}_{0 \le i < n_{exp}}$
(20) **(Typing 2)** $\{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathsf{parseUExp}(\mathsf{subseq}(b; m''_i; n''_i)) \rightsquigarrow e_i : [\omega]\tau_i\}_{0 \le i < n_{exp}}$ *and* $\{\Omega_{app} \vdash e_i : [\omega]\tau_i\}_{0 \le i < n_{exp}}$
(21) **(Capture Avoidance)** $e = [\{\kappa_i/k_i\}_{0 \le i < n_{kind}}, \{c_i/u_i\}_{0 \le i < n_{con}}, \{e_i/x_i\}_{0 \le i < n_{exp}}, \omega]e''$ *for some* e'' *and fresh* $\{k_i\}_{0 \le i < n_{kind}}$ *and fresh* $\{u_i\}_{0 \le i < n_{con}}$ *and fresh* $\{x_i\}_{0 \le i < n_{exp}}$
(22) **(Context Independence)** $\mathsf{fv}(e'') \subset \{k_i\}_{0 \le i < n_{kind}} \cup \{u_i\}_{0 \le i < n_{con}} \cup \{x_i\}_{0 \le i < n_{exp}} \cup dom(\Omega_{params})$

8 Related Work

8.1 Syntax Definition Systems

One approach available to library providers seeking to introduce new literal forms is to use a syntax definition system to construct a library-specific *syntax dialect*: a new syntax definition that extends the syntax definition given in the language definition with new forms, including literal forms.

There are hundreds of syntax definition systems of various design. Notable examples include grammar-oriented systems like Camlp4 [37], Copper [60, 69] and Sugar*/SoundExt [17, 18, 39, 40], as well as parser combinator systems [31]. 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 build system (e.g. `ocamlbuild`), or via language-integrated preprocessing directives, e.g. Racket's `#lang` directive or its reader macros [20], or the import mechanism of SugarJ [17].

The problem was described by example in Sec. 1: these systems make it difficult to reason abstractly. Let us reitierate more generally, before discussing a few systems that are exceptional along some strict subset of these dimensions:

(1) **Responsibility**: Clients using a combined syntax dialect cannot easily determine which constituent extension is responsible for a given form, whereas TLMs have explicit names which follow the usual scoping conventions.
Moreover, there can be syntactic conflicts because multiple extensions can claim responsibility for the same form. TLMs sidestep these complexities entirely because the context-free syntax of the language is fixed, and composition is via splicing rather than direct combination.
(2) **Segmentation**: Clients of a syntax dialect cannot accurately determine which segments of the program text appear directly in the desugaring. In contrast, TLM clients can inspect the inferred segmentation (communicated via secondary notation.)
(3) **Capture**: Unlike TLM clients, clients of a syntax dialect cannot be sure that spliced terms are capture avoiding.
(4) **Context Dependence**: Similarly, clients cannot be sure that the desugaring is context independent. Indeed, without a method to pass in parameters (Sec. 4), achieving strict context independence would be impractical.
(5) **Typing**: Clients of a syntax dialect cannot reason abstractly about what type a desugaring has. In contrast, TLM clients can determine the type of any expansion by referring to the parameter and type declarations on the TLM definition, and nothing else. The inferred segmentation also gives types for each spliced expression or pattern.

An extensible syntax definition system that has confronted the problem of **Responsibility** (but not the other problems) is Copper [60]. Copper integrates a modular grammar analysis that guarantees that determinism is conserved 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 the domain name system or some other coordinating entity. 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 be abbreviated (Sec. 4).

Some programming languages, notably including theorem provers like Coq [44] and Agda [51], support "mixfix" notation directives [24, 49, 71]. Many of these systems enforce capture avoidance and application-site context independence [13, 24, 44, 65]. The problem is that mixfix notation requires a fixed number of sub-trees, e.g. `if _ then _ else _`. Coq has some limited extensions for list-like literals [44]. These systems cannot express the example literal forms from this paper, because they can have any number of spliced terms.

The work of Lorenzen and Erdweg [39, 40] introduces SoundExt, 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. TLMs differ in several ways. First, as already discussed, we leave the context-free syntax fixed, so different TLMs cannot conflict. Second, SoundExt does not enforce hygiene, i.e. expansions might depend on the context and intentionally induce capture. Similarly, there is no abstract segmentation discipline. A client can only indirectly reason about binding (but not segmentation) by inspecting the derived typing rules. Unlike TLMs, SoundExt supports type-dependent expansions [40]. The trade-off is that TLMs can generate expansions, and therefore segmentations, even when the program is ill-typed. Another important distinction is that TLMs rely on proto-expansion validation, rather than verification as in SoundExt. The trade-off is that TLMs do not require that the expansion logic be written using a restricted rewriting system, nor does the system require a fully mechanized language definition. Finally, there is no clear notion of "partial application" in SoundExt or other syntax definition systems.

8.2 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.

OCaml's textual syntax now includes *preprocessor extension (ppx) points* used to identify terms that some external term rewriting preprocessor must rewrite [37]. We could mark a string literal as follows:

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

More than one applied preprocessor might recognize this annotation (there are, in practice, many XML/HTML libraries), so the problems of **Responsibility** comes up. It is also impossible to reason abstractly about **Segmentation**, **Capture**, **Context Dependence** and **Typing** because the code that the preprocessor generates is unconstrained.

Term-rewriting macro systems are language-integrated local term rewriting systems that require that the client explicitly apply the intended rewriting, implemented by a macro, to the term that is to be rewritten. This addresses the issue of **Responsibility**. However, unhygienic, untyped macro systems, like the earliest variants of the Lisp macro system [28], Template Haskell [62] and GHC's quasiquotation system [43] (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. (It is not enough that with Template Haskell / GHC quasiquotation, the generated expansion is typechecked – to satisfy the **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 [29, 30], **Capture**, and they enforce application-site **Context Independence** [3, 11, 16, 35]. The critical problem is that a standard hygiene discipline makes it impossible to repurpose string literal forms to introduce compositional literal forms at other types. Consider again our running XHTML example, which we might try to realize by applying a hygienic macro, `xml!`, to a string literal that the macro parses:

```
(xml! "SSTR<h1>Hello, {[first_name]}!</h1>ESTR")
```

The expansion of this macro will fail the check for application-site context independence because `first_name` will appear as a free variable, in no way different from any other. The hygiene mechanism for TLMs addresses this problem by explicitly distinguishing spliced segments of the literal body in the proto-expansion. This also addresses the problem of **Segmentation** – the segmentation abstractly communicates the fact that (only) `first_name` is a spliced **`string`** expression.

Much of the research on macro systems has been for languages in the LISP tradition [45] that do not have rich static type structure. The formal macro calculus studied by Herman and Wand [30] (which is not capable of expressing new literal forms, for the reasons just discussed) uses types only to encode the binding structure of the generated expansion (as discussed in Sec. 2.4.1). Research on typed *staging macro systems* like MetaML [61], MetaOCaml [34] and MacroML [21] is also not directly applicable to the problem of defining new literal forms – the syntax tree of the arguments cannot be inspected at all (staging macros are used mainly for partial evaluation and performance-related reasons.)

The Scala macro system is a hygienic macro system. Its "black box" macros support reasoning abstractly about **Typing** because type annotations constrain the macro arguments and the generated expansions, though the precise reasoning principles available are unclear because the Scala macro system has not been formally specified. The full calculus we have defined is the first detailed type-theoretic accounts of a typed, hygienic macro system of any design for an ML-like language, i.e. one with a rich static type system, support for pattern matching, type functions and ML-like modules.

Some languages, including Scala [52], build in *string splicing* (a.k.a. *string interpolation*) forms, or similar but more general *fragmentary quotation forms* [63], e.g. SML/NJ. 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. In the HTML example, we have splicing at two different types using two different delimiters. These forms also cannot appear in patterns.

This brings us back to the most closely related work, that of Omar et al. [53] 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 form. 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 can operate at any type, including structural types. In a subsequent short paper, Omar et al. [54] suggested explicit application of simple expression TLMs also in a bidirectional typed setting [56], but this paper did not have any formal content. With TLMs, it is not necessary for the language to be bidirectionally typed (see Sec. 2.4.3 on type inference).

Perhaps most importantly, the metatheory presented by Omar et al. [53] establishes only that generated expansions are of the expected type (i.e. a variant of the Typed Expression Expansion theorem from Sec. 6.7.) It does not establish the remaining abstract reasoning principles that have been the major focus of this paper. In particular, 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 segments.

Finally, the prior work did not consider pattern matching, type functions, ML-style modules, parameters or static evaluation. This paper addresses all of these.

9 Discussion

The importance of specialized notation as a "tool for thought" has long been recognized [32]. According to Whitehead, a good notation "relieves the brain of unnecessary work" and "sets it free to concentrate on more advanced problems" [8], 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 [59], "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 case studies detailed in this paper and in the supplement, and the examples collected from the empirical study by Omar et al. [53].

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 analagous to those already familiar to programmers in languages like ML and Scala.

A correct parse function never returns an encoding of a proto-expansion that fails validation given well-typed splices, but this invariant cannot be enforced by the ML type system. Under a richer type system, the return type of the parse function itself could be refined so as to enforce this invariant intrinsically. This problem – of typed first-class typed term representations – has been studied in a variety of settings, e.g. in MetaML [61] and in the modal logic tradition [14]. Our present efforts aim to leave the semantics of OCaml unchanged.

Another 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. [58] 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 [22] remain to be considered, and indeed these are our biggest remaining implementation challenges.

REFERENCES

[1] [n. d.]. Information Technology Portable Operating System Interface (POSIX) Base Specifications, Issue 7. ([n. d.]).

[2] [n. d.]. Merlin, an assistant for editing OCaml code. https://the-lambda-church.github.io/merlin/. Retrieved Sep 6, 2016. ([n. d.]).

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

[4] 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.

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

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

[7] 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.

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

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

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

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

[12] Karl Crary. 2009. A syntactic account of singleton types via hereditary substitution. In *Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)*. http://doi.acm.org/10.1145/1577824.1577829

[13] 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

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

[15] Derek Dreyer. 2005. *Understanding and evolving the ML module system*. Ph.D. Dissertation. Carnegie Mellon University.

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

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

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

[19] 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.

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

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

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

[23] 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.

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

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

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

[27] Robert Harper, John C Mitchell, and Eugenio Moggi. 1989. Higher-order modules and the phase distinction. In *POPL*.

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

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

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

[31] 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

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

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

[34] 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

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

[36] Daniel K. Lee, Karl Crary, and Robert Harper. 2007. Towards a mechanized metatheory of Standard ML. In *POPL*. http://dl.acm.org/citation.cfm?id=1190216

[37] 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.

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

[39] 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

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

[41] David MacQueen. 1984. Modules for Standard ML. In *Symposium on LISP and Functional Programming*. http://doi.acm.org/10.1145/800055.802036

[42] David B. MacQueen. 1986. Using Dependent Types to Express Modular Structure. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986.* 277–286. https://doi.org/10.1145/512644.512670

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

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

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

[46] 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.

[47] Heather Miller, Philipp Haller, and Martin Odersky. 2014. Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings.* 308–333. https://doi.org/10.1007/978-3-662-44202-9_13

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

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

[50] Enrico Moriconi and Laura Tesconi. 2008. On inversion principles. *History and Philosophy of Logic* 29, 2 (2008), 103–113.

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

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

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

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

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

[56] 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

[57] Gordon D. Plotkin. 2004. A structural approach to operational semantics. *J. Log. Algebr. Program.* 60-61 (2004), 17–139.

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

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

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

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

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

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

[64] Christopher A Stone and Robert Harper. 2006. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic (TOCL)* 7, 4 (2006), 676–722.

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

[66] The Reason Team. 2017. Reason Guide: What and Why? (2017). https://reasonml.github.io/guide/what-and-why/. Retrieved Nov. 14, 2017.

[67] 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

[68] 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

[69] Eric Van Wyk and August Schwerdfeger. 2007. Context-aware scanning for parsing extensible languages. In *GPCE*. http://doi.acm.org/10.1145/1289971.1289983

[70] 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

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

[72] Jeremy Yallop and Leo White. 2015. Modular macros (extended abstract). In *OCaml Users and Developers Workshop*.