# Reasonably Programmable Literal Forms

ANONYMOUS AUTHOR(S)

General-purpose programming languages typically provide literal notation for only a small number of standard data structures, like lists. This common design practice is decidedly *ad hoc*: there are many other data structures for which literal notation might also be useful, e.g. finite maps, regexes, HTML data, SQL queries, syntax trees and chemical structures. Availability of literal notation is far from a superficial concern: studies suggest that literal notation can have a serious impact on software quality by subverting string encodings, which are vectors for injection attacks and cause other difficulties. This paper develops, from type-theoretic first principles, a mechanism that decentralizes control over literal notation: *typed literal macros (TLMs)*. TLMs give library providers programmatic control over the parsing and expansion of expressions and patterns of a flexible *generalized literal form* at a specified type or parameteric family of types. Compared to existing approaches, including syntax extension systems and macro-based quotation systems, TLMs are uniquely *reasonable*: TLM providers can reason modularly about syntactic determinism, and TLM clients can reason abstractly about the type and binding structure of the program. The system only needs to convey to clients (via secondary notation) the inferred *segmentation* of each literal, which gives the locations of spliced terms. Maintaining information about segmentation is necessary for the novel TLM hygiene mechanism, i.e. the mechanism that ensures that clients can reason abstractly about binding (previously, only unhygienic macro systems supported splicing of terms out of string literal bodies.) This paper establishes these abstract reasoning principles formally with a calculus of typed expressions, patterns and modules. This calculus is the first detailed type-theoretic account of a hygienic term-rewriting 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 a common practice to build in *literal forms* that decrease the syntactic cost of constructing and pattern matching over values of a select few standard types. For example, in Standard ML (SML) and its cousins, a list expression literal like `[x, y, z]` is equivalent by desugaring to `Cons(x, Cons(y, Cons(z, Nil)))` where `Nil` and `Cons` stand for the constructors of the standard `list` datatype (Harper 1997; Milner et al. 1997).[1] Another example is found in Ur/Web's surface syntax, which extends Ur's surface syntax with expression and pattern literal forms for encodings of HTML data (Chlipala 2010, 2015). Fig. 1 shows two such HTML literals, one that "splices in" a string expression (Line 1) and the other an HTML expression (Line 2).

This common language design practice is decidedly *ad hoc* – it is quite easy to come up with other examples of data structures that are not privileged with literal forms in these languages, but for which mathematicians, scientists or programmers have invented specialized notation (Cajori 1928; Iverson 1980). Consider, for example, that 1) clients of a "collections" library might like not just list literals, but also matrix, set and map literals following standard mathematical conventions; 2) clients of a "web programming" library might like not just HTML literals but also CSS literals (which Ur/Web does not have); 4) a compiler writer might like "quotation" literals for the terms of the object language and various intermediate languages; and 5) clients of a "chemistry" library might like chemical structure literals based on the SMILES standard (Anderson et al. 1987).

[1] In SML, list literals work even in contexts where the list expression constructors have been shadowed by other bindings, i.e. the meaning of literal forms is *context independent*. We will return to the concept of context independence again.

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

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

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

Although concerns about "missing" literal notation are easy to dismiss as superficial or frivolous, the reality is that literal forms, or the absence thereof, have a substantial influence on software quality. For example, Bravenboer et al. (2007) finds that literal forms for structured encodings of queries, like the SQL-based query literals now found in many languages (Meijer et al. 2006), reduce the temptation to use string encodings of queries and therefore reduce the prevalence of string injection attacks, which are amongst the most common and serious security vulnerabilities on web servers today (OWASP 2017). More generally, evidence suggests that programmers frequently engage in "stringly-typed programming", i.e. they choose strings instead of composite data structures largely for reasons of syntactic convenience. In particular, Omar et al. (2014) sampled strings from open source projects and found that about 15% of them were parseable according to some readily apparent type-specific grammar, e.g. for SQL queries, regexes, file system paths, URLs and many other data structures. Literal forms, with support for splicing, would decrease the syntactic cost of composite encodings of such data, which are more amenable to programmatic manipulation and compositional reasoning than flat string encodings.

Of course, it would not be sensible to ask general-purpose language designers to accomodate all known notations with built-in literal forms. Instead, there has been considerable interest in mechanisms that allow library providers to define new literal notation on their own. For example, general syntax extension mechanisms, including Camlp4 (Leroy et al. 2014), SugarJ/Sugar* (Erdweg et al. 2011; Erdweg and Rieger 2013) and others that we will return to in Sec. 7, can be used to define new literal forms. The problem that specifically motivates this paper is that these mechanisms can obscure program structure and are therefore *unreasonable* for "programming in the large" (DeRemer and Kron 1976). Consider, for instance, the perspective of a programmer attempting to comprehend (i.e. reason about) the program text in Fig. 2, which is written in a dialect of OCaml's syntax called Reason[2] that has, hypothetically, been extended with a large number of new literal forms.

```
1 let w = compute_w();
2 let x = compute_x(w);
3 let y = [|(!R)@&{&/x!/:2_!x}'!R}|];
```

Fig. 2. Syntax extension systems produce unreasonable program text.

The literal form on Line 3 constructs an encoding of a query in the niche stack-based database query language K, using its terse surface syntax (Whitney and Shasha 2001). Of course, the programmer reading this program text may be unfamiliar with K's query syntax. Unfortunately for this programmer, there are no simple, abstract protocols for answering questions like:

(1) **(Responsibility)** Which syntax extension determined the desugaring of the literal form on Line 3? Might there be many possible desugarings (i.e. syntactic non-determinism)?
(2) **(Segmentation)** Are the characters `x` and `R` on Line 3 taken as spliced expressions (meaning that they appear directly in the desugaring), or are they parsed in some other way peculiar to this literal form (e.g. as operators in the K query language)?
(3) **(Capture)** If `x` is in fact a spliced expression, does it refer to the binding of `x` on Line 2, or might it capture an unseen binding of the same identifier in the desugaring of Line 3?
(4) **(Context Dependence)** If `w`, on Lines 1-2, is renamed, could that possibly break Line 3 or change its meaning because the desugaring assumes that some variable `w` is in scope?
(5) **(Typing)** What type does `y` have?

[2]http://facebook.github.io/reason

```
1 let w = compute_w();
2 let x = compute_x(w);
3 let y = $kquery [|(!R)@&{&/x!/:2_!x}'!R}|];
```

Fig. 3. TLMs make examples like the one from Fig. 2 more reasonable.

Forcing the programmer to reason transparently, i.e. to examine the desugaring of the program text, or even the logic that produced that desugaring, to answer basic questions like these defeats the ultimate purpose of syntactic sugar: decreasing cognitive cost (Green 1989). Analagous problems do not arise when programming without syntax extensions in languages like ML and Scala – the binding discipline of the language ensures that programmers can reason lexically about binding, and types mediate abstraction over function and module implementations.

This paper formally details a more reasonable mechanism for defining new literal forms: *typed literal macros (TLMs)*. TLMs give library providers full programmatic control over the parsing and expansion of *generalized literal forms* (Omar et al. 2014), which syntactically subsume other literal forms because their bodies are left initially unparsed according to the fixed context-free syntax of the language (i.e. they are initially parsed like raw string literals.) For example, Fig. 3 applies a TLM, `$kquery`, to a generalized literal form delimited by `[|` and `|]` to express the "unreasonable" K query example from Fig. 2. The applied TLM parses the literal body and generates an expansion statically, during a semantic phase called typed expansion, which generalizes the usual typing phase. TLMs come equipped with uniquely powerful abstract reasoning principles. We will characterize these reasoning principles precisely as we proceed, but for now, let us develop the basic intuitions – the programmer can reason abstractly about Line 3 as follows:

(1) **(Responsibility)** The applied TLM, `$kquery`, parses and determines the expansion of the literal body. The context-free syntax of the language is fixed, so the TLM provider is able to modularly establish that the syntax this TLM implements is deterministic.
(2) **(Segmentation)** The output that the TLM generates is structured so that the system can infer from it an accurate *segmentation* of the literal body that distinguishes spliced terms from segments that are parsed in some other way by the TLM. The segmentation can be communicated by a code editor using secondary notation, e.g. colors in this document. So by examining Fig. 3, the programmer knows that the two instances of `x` on Line 3 are parsed as spliced expressions (because they are in black), whereas the `R`'s must be parsed in some other way, e.g. as operators in the K query language (because they are in green.)
(3) **(Capture)** Splicing is capture-avoiding, so the spliced expression `x` must refer to the binding of `x` on Line 2. It cannot capture a coincidental binding of `x` in the expansion.
(4) **(Context Dependence)** The system enforces context independence, so the expansion of Line 3 cannot rely on the fact that, e.g., `w` is in scope. The client is free to rename `w`.
(5) **(Typing)** An explicit type annotation on the definition of `$kquery` determines the type that the expansion must have. We will see an example of a TLM definition in the next section. Moreover, each spliced segment in the segmentation also comes with a type annotation.

The remainder of this document is organized as follows. Sec. 2 defines another simple expression TLM – one that implements the Ur/Web-style HTML literal syntax from Fig. 1 – and uses it as a running example to informally describe how TLMs work from start to finish. Sec. 3 then briefly introduces pattern TLMs, i.e. TLMs that generate patterns. Sec. 4 defines a type-theoretic calculus of simple expression and pattern TLMs and formally establishes the reasoning principles implied above. Sec. 5 then adds type functions and an ML-style module system and introduces *parametric TLMs* (pTLMs), i.e. TLMs that can take type and module parameters. Parameters serve two important purposes. First, they allow a TLM to operate not just at a single type, but over a type- and module-parameterized family of types. We give as an example a parametric TLM for

string-keyed dictionaries that operates uniformly over different implementations of the abstract dictionary signature, and also over different choices of the range type. Second, parameters lower the syntactic cost of the strict context independence discipline that TLMs enforce because they give expansions access to terms other than those explicitly spliced in to the literal body by the client, and because partial parameter application is possible. Sec. 6 considers the topic of term evaluation during the typed expansion phase, i.e. *static evaluation*, in more detail, and also gives examples of TLMs that are useful for defining other TLMs, e.g. TLMs that implement parser generators and quasiquotation. Sec. 7 compares TLMs to related work, including recent related work by Omar et al. (2014, 2015), and the long history of related work on macro systems. Sec. 8 concludes with a summary of the specific contributions of this paper and a brief discussion of the limitations of TLMs, along with some suggestions for future work.

Due to space limitations, certain technical details of the calculi described in this paper, as well as detailed proofs, are given in the supplemental material. The examples in the remainder of the paper are written in an open source alternative syntactic front-end for OCaml called Reason, mentioned above, because we are working to incorporate TLMs into Reason. However, the essential ideas are not Reason-, OCaml- or ML-specific – it should be possible to adapt TLMs to other languages that take a similar approach to types and binding (e.g. Haskell, Scala and others.) Languages that have a disciplined binding structure but that lack a rich static type structure, e.g. standard Racket, would also benefit from TLMs – they can apply the results of this paper by deploying the usual trick of viewing the language as statically "unityped" (Harper 2016; Scott 1980).

## 2 Simple Expression TLMs (seTLMs)

We will start in this section by informally describing TLMs in Reason. It will suffice in this section to consider only *simple expression TLMs (seTLMs)*, i.e. TLMs for abbreviating expressions of a single specified type. In particular, we will consider an seTLM named `$html` that implements Ur/Web style HTML literal notation for expressions of the `html` type that is defined in Fig. 4.

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

Fig. 4. The `html` type, which classifies encodings of HTML data. The type `html_attr` is also straightforward and the type function `list` is standard, so neither are shown. Strings are primitive.

### 2.1 TLM Definition and Application

The definition of the `$html` TLM is outlined in Fig. 5. TLM names are prefixed by `$` to distinguish them from variable identifiers. Every TLM definition also has a *type annotation*, here **`at`** `html`, and a *parse function* between **`by static`** `{` and `}`. The TLM definition is in scope until the end of the enclosing scope. We will describe how TLM definitions are packaged into libraries in Sec 6.

```
1 syntax $html at html by static {
2   fun(b : body) : parse_result(proto_expr) =>
3     /* ... Ur/Web-style HTML syntax parser ... */
4 };
```

Fig. 5. The definition of the seTLM named `$html`.

Using the TLM defined in Fig. 5, we can express the example Ur/Web program from Fig. 1 as shown in Fig. 6. On both lines, we apply `$html` to a *generalized literal form* delimited by `[|` and `|]`. Generalized literal forms, which first arose in related work by Omar et al. (2014) that we will detail in Sec. 7, 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. (2014) formally specified several other choices, including layout-sensitive delimitation.) *Literal bodies*, i.e. the characters between these delimiters, are constrained only in that occurrences of `[|` must be balanced by `|]`.

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

Fig. 6. Two examples of the `$html` TLM being applied.

The semantics delegates responsibility over parsing and expansion of each literal body to the applied TLM's parse function during a semantic phase called *typed expansion*, which generalizes the usual typing phase. Because the parse function is applied during the typed expansion process, 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.) For now, we will assume that static functions are completely closed, and that they do not themselves make use of TLMs (we will eliminate these impractical limitations by introducing distinct static phase bindings in Sec. 6.)

The parse function must have type `body -> parse_result(proto_expr)`. The input type, `body`, classifies encodings of literal bodies. Literal bodies are sequences of characters, so it suffices to define `body` as an abbreviation for **`string`**, as shown in Fig. 7. The return type is a labeled sum type, defined by applying the type function `parse_result` defined in Fig. 7, that distinguishes between parse errors and successful parses.[3] Let us consider these two possibilities in turn.

If the parse function determines that the literal body is not well-formed according to whatever syntax definition that it implements, it must return `ParseError {msg=`$e_{\text{msg}}$`, loc=`$e_{\text{loc}}$`}` where $e_{\text{msg}}$ is a custom error message and $e_{\text{loc}}$ is a value of type `segment`, defined in Fig. 7, that designates a segment of the literal body as the origin of the error (van Deursen et al. 1993). This information is reported to the programmer by the Reason compiler or editor services.

If, on the other hand, parsing succeeds, the parse function returns `Success` $e_{\text{proto}}$, where $e_{\text{proto}}$ is called the *encoding of the proto-expansion*. For expression TLMs, the proto-expansions is a *proto-expression*. These are encoded as values of the recursive sum type `proto_expr` that is outlined in Fig. 7.[4] Most of the constructors of `proto_expr` are individually uninteresting – they encode Reason's various expression forms (just as in a self-hosted compiler, e.g. SML/NJ's Visible Compiler library.) Expressions can mention types, so we also need the type `proto_typ` also outlined in Fig. 7. It is only the `SplicedE` and `SplicedT` constructors that are novel. These are discussed next.

## 2.2 Splicing

When the parse function determines that some segment of the literal body should be taken as a spliced expression, according to whatever syntactic criteria it deems suitable, it must not directly insert the syntax tree of that expression into the encoding of the expansion. Instead, the TLM must indirectly refer to the spliced expression using the `SplicedE` variant 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` variant 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` variant of `proto_typ` analagously.

For example, consider again the TLM application on Line 1 of Fig. 6, reproduced below:

```
$html [|<h1>Hello, {[first_name]}</h1>|]
```

The proto-expansion generated by `$html`, if written in a textual syntax for proto-expressions where references to spliced expressions are written **`spliced`**`<startIdx; endIndex; ty>`, is:

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

[3] We might alternatively have used an exception to signal a parse error, but sum types are formally simpler.

[4] More sophisticated encoding techniques for syntax trees would cause no fundamental problems; we have kept it simple here for the sake of clarity.

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

Fig. 7. Definitions of various types used by TLM definitions. We assume that these are available ambiently.

The proto-expression **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. It corresponds to the following 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 below, 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 is the finite set containing only **spliced**<13; 22; **string**>. The semantics 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-typed program has a well-defined segmentation. A program editor or pretty-printer can communicate this location 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, if strings were not primitive but rather defined as sequences of characters, we might define a TLM `$str` to recover string literal notation and write `$html [|<body>{heading ($str [|World!|])} ...</body>|]`.

A program editor or pretty-printer can also communicate the type of each spliced expression, as specified by the segmentation, to the programmer upon request (the editor packages for working with Reason defer to the Merlin tool when the programmer requests the type of an expression.)

### 2.3 Proto-Expansion Validation

Three important concerns described in Sec. 1 remain: those related to reasoning abstractly about **Capture**, **Context Dependence** and **Typing**. Addressing these concerns is the purpose of the *proto-expansion validation* process, which occurs once a proto-expansion has been generated by the parse function. 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.

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

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

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

```
let tmp = /* ... expansion-internal temporary ... */;
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. For example, the final expansion of the example above might take the following form:

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

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

For TLM providers, the benefit of this mechanism is that they can name the variables used internally within expansions freely. There is no need to explicitly deploy a mechanism that generates "fresh variables". TLM clients can, in turn, can reason abstractly about capture, i.e. without examining the expansion that the spliced term appears within.

This does prevent library providers from intentionally introducing bindings into spliced terms. For example, Haskell's literal notation for monadic commands, i.e. **do**-notation, supports binding the result of executing a command to a variable that is then available in the subsequent commands in the command sequence. Using TLMs in Reason, this notation cannot be expressed – values can be communicated from the expansion to a spliced expression only via function arguments. It would be possible to distinguish spliced identifiers, much as we do spliced terms, and explicitly make them available to spliced expressions by extending the `SplicedE` and `SplicedT` constructors to take finite sets of spliced identifiers (following the basic approach taken by Herman and Wand (2008).) However, the opinion of the authors is that this would unjustifiably increase the reasoning burden on clients when they encounter an unfamiliar literal form: *might this new literal form be doing obscure things with binding too?* Haskell-style infix notation for monadic commands, where bind is `e >>= f`, can be expressed using TLMs and splicing.

*2.3.2 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 never generates context-independent proto-expansions is below:

```
syntax $broken at rx by static { fun(_) => Success (Var "x") };
```

The proto-expansion that this TLM generates (for any given 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, so this proto-expansion is deemed invalid (even when `$broken` is applied where `x` is coincidentally bound.)

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. Indeed, we saw examples of spliced terms that referred to variables bound at the application site above. Because proto-expansions refer to spliced terms indirectly, enforcing context independence is straightforward – we need only that the proto-expansion itself be closed.

One subtlety is that we assumed in the examples above that constructors like `H1Element` were available to the proto-expansion. In OCaml and other dialects of ML where datatype constructors are injected into the context as variables when the datatype is defined, this would violate context independence. As such, we will have to assume for now and in our formalism in Sec. 4 that explicit injections into labeled sum types are available. Once we introduce module parameters in Sec. 5, we will be able to pass in a module exporting the datatype constructors as a parameter and partially apply that parameter to hide this detail from clients.

*2.3.3 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.

## 3 Simple Pattern TLMs (spTLMs)

Let us now briefly consider the topic of TLMs that expand to *patterns*, rather than expressions. For example, we can pattern match on a value `x : html` by applying a pattern TLM `$html` as follows:

```
let children_of_h1 x =
  switch x {
  | $html [|<h1>{cs}</h1>|] -> Some cs; /* 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.

Pattern TLM definitions look much like expression TLM definitions:

```
syntax $html at html for patterns by static {
  fun(b : body) : parse_result(proto_pat) =>
    /* ... Ur/Web-style HTML pattern parser here ...  */
};
```

The *sort qualifier*, **`for patterns`**, indicates that this is a pattern TLM definition. Expression TLM definitions can also include a sort qualifier, **`for expressions`**, but this is the default if omitted. Notice that we used the same name, `$html`, for this pattern TLM as for the expression TLM defined in the previous section. This is possible because patterns and expressions are distinct sorts of terms. It does not make sense to apply an expression TLM in pattern position, and *vice versa*.[5] The return type of the parse function is `parse_result(proto_pat)`, rather than `parse_result(proto_expr)`. The type `proto_pat`, outlined in Fig. 8, classifies encodings of *proto-patterns*.

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

Fig. 8. Abbreviated definition of `proto_pat`.

The constructor `SplicedP` serves much like `SplicedE` to allow a proto-pattern to refer indirectly to spliced pattern segments by their location within the literal body. For example, the proto-pattern generated for the example at the top of this section would be written concretely as follows:

```
H1Element (_, spliced<6; 7; list(html)>)
```

[5] The fact that some patterns look like expressions in the textual syntax is immaterial to this fundamental semantic distinction. Many expression-level constructs, e.g. lambdas, do not correspond even syntactically to patterns.

To maintain a reasonable abstract binding discipline, i.e. to allow clients to reason about variable binding without examining pattern TLM expansions directly, 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 (though wildcards are allowed.) This prohibition ensures that no variables other than those visible to the client in a spliced pattern are captured by the corresponding branch of the match expression.

Patterns have no way to refer to surrounding bindings (they only induce bindings in other expressions, e.g. in the corresponding branch of the **`switch`** expression.) However, 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 a reasonable abstract typing discipline, proto-pattern validation checks:

(1) that the final expansion is a pattern that matches values of the type specified by the TLM's type annotation;
(2) that each spliced pattern matches values of the type indicated in the segmentation; and
(3) that each of these types are themselves well-formed types.

## 4 Simple TLMs, Formally

Before continuing on to consider more general parametric TLMs, let us develop a calculus of simple expression and pattern TLMs called $\lambda^{\mathbf{S}}_{\mathbf{TLM}}$. This calculus consists of an *unexpanded language*, or *UL*, defined by typed expansion to an *expanded language*, or *XL*. Figs. 9 and 10 summarize the syntax of the UL and the XL, respectively. Programs are written as unexpanded expressions but evaluate as well-typed expanded expressions. We will start with a brief overview of our XL before turning in the remainder of the section on the UL and the typed expansion process.

### 4.1 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. The reader is directed to *PFPL* (Harper 2012) for a detailed introductory account of these standard constructs. We will only tersely summarize the statics and dynamics of the XL below because the particularities of the XL are not critical to the ideas we will introduce below.

The *statics of the XL* is defined by hypothetical judgements of the following form:

| | |
|---|---|
| $\Delta \vdash \tau$ type | $\tau$ is a well-formed type |
| $\Delta\ \Gamma \vdash e : \tau$ | $e$ is assigned type $\tau$ |
| $\Delta\ \Gamma \vdash r : \tau \Longmapsto \tau'$ | $r$ takes values of type $\tau$ to values of type $\tau'$ |
| $\Delta \vdash p : \tau \dashv \Gamma$ | $p$ matches values of type $\tau$ and generates hypotheses $\Gamma$ |

*Type formation contexts*, $\Delta$, are finite sets of hypotheses of the form $t$ type. *Typing contexts*, $\Gamma$, are finite functions that map each variable $x \in \mathrm{dom}(\Gamma)$, where $\mathrm{dom}(\Gamma)$ is a finite set of variables, to the hypothesis $x : \tau$, for some $\tau$. The judgements above are inductively defined in the supplemental material and validate standard lemmas, also given in the supplement.

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 $e'$. Additional judgements, not shown, are needed to define the dynamics of pattern matching, but they do not appear directly in our subsequent developments, so we omit them. We assume an eager dynamics and the standard type safety theorem (see supplement.)

### 4.2 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,

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

Fig. 9. Syntax of the $\lambda^{\mathbf{S}}_{\mathbf{TLM}}$ unexpanded language (UL). Metavariable $\hat{t}$ ranges over type identifiers, $\hat{x}$ over expression identifiers, $\ell$ 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.

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

Fig. 10. Syntax of the $\lambda^{\mathbf{S}}_{\mathbf{TLM}}$ expanded language (XL). XL terms are *abstract binding trees* (ABTs) identified up to alpha-equivalence, so we follow the syntactic conventions of Harper (2012). Metavariable $x$ ranges over variables and $t$ over type variables.

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, and we return to it below.

Most of the unexpanded forms in Figure 9 mirror the expanded forms. We refer to these as the *common forms*. The mapping from expanded forms to common unexpanded forms is defined explicitly in the supplement.

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 need 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 necessary condition in full.)

### 4.3 Typed Expansion

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

| | |
|---|---|
| $\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau$ type | $\hat{\tau}$ has well-formed expansion $\tau$ |
| $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau$ | $\hat{e}$ has expansion $e$ of type $\tau$ |
| $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \rightsquigarrow r : \tau \Longmapsto \tau'$ | $\hat{r}$ has expansion $r$ taking values of type $\tau$ to values of type $\tau'$ |
| $\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\Vdash \hat{\Gamma}$ | $\hat{p}$ has expansion $p$ matching against $\tau$ generating hypotheses $\hat{\Gamma}$ |

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 below, and the remainder are in the supplement:

$$\textsc{ee-id}\ \frac{}{\hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{x} \rightsquigarrow x : \tau}$$

$$\textsc{ee-lam}\ \frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \text{type} \qquad \hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \lambda \hat{x}{:}\hat{\tau}.\hat{e} \rightsquigarrow \texttt{lam}\{\tau\}(x.e) : \texttt{parr}(\tau;\tau')}$$

The only subtlety here 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 $\mathcal{G}$ is an *expression identifier expansion context*, and $\Gamma$ is a standard typing context. An expression identifier expansion context, $\mathcal{G}$, is a finite function that maps each expression identifier $\hat{x} \in \mathrm{dom}(\mathcal{G})$ to the hypothesis $\hat{x} \rightsquigarrow x$, for some expression variable, $x$, called its expansion. We write $\mathcal{G} \uplus \hat{x} \rightsquigarrow x$ for the expression identifier expansion context that maps $\hat{x}$ to $\hat{x} \rightsquigarrow x$ 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 $\hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau$ when $\hat{\Gamma} = \langle \mathcal{G}; \Gamma \rangle$ as an abbreviation of $\langle \mathcal{G} \uplus \hat{x} \rightsquigarrow x; \Gamma, x : \tau \rangle$.

To develop an intuition for why the update operation is necessary, it is instructive to inspect the derivation of the expansion of the unexpanded expression $\lambda\hat{x}{:}\hat{\tau}.\lambda\hat{x}{:}\hat{\tau}.\hat{x}$ to $\texttt{lam}\{\tau\}(x_1.\texttt{lam}\{\tau\}(x_2.x_2))$ assuming $\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau$ type:

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

Notice that when Rule EE-LAM is applied, the type identifier expansion context is updated but the typing context is extended with a (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.

*Unexpanded type formation contexts*, $\hat{\Delta}$, consist of a *type identifier expansion context*, $\mathcal{D}$, paired with a standard type formation context, $\Delta$, and operate analagously (see supplement.)

Before we continue, let us state an important invariant: that typed expression expansion produces a well-typed expression.

Theorem 4.1 (Typed Expression Expansion). *If* $\langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau$ *then* $\Delta\ \Gamma \vdash e : \tau$.

For the typed expansion rules governing common forms, like the two example rules above, the typed expansion rules mirror the corresponding typing rules so it is easy to see that this invariant holds. The details are in the supplement. The rules of particular interest are the rules governing TLM definitions and TLM application, which are the topic of the remainder of this section.

### 4.4 TLM Definitions

The rule below defines typed expansion of the seTLM definition form:

$$\textsc{ee-def-setsm}\quad \dfrac{\begin{array}{ll}\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \text{ type} & \emptyset\ \emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResultSE)} \\ e_{\text{parse}} \Downarrow e'_{\text{parse}} & \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}, \hat{a} \rightsquigarrow a \hookleftarrow \texttt{setlm}(\tau; e'_{\text{parse}}); \hat{\Phi}} \hat{e} \rightsquigarrow e : \tau'\end{array}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for expressions by static}\ e_{\text{parse}}\ \texttt{in}\ \hat{e} \rightsquigarrow e : \tau'}$$

The first premise expands the unexpanded type annotation. The second premise checks that $e_{\text{parse}}$ is a closed expanded function[6] of type `parr(Body;ParseResultSE)`.

The type abbreviated `Body` classifies encodings of literal bodies, $b$. The mapping from literal bodies, $b$, to values of type `Body` is defined by the *body encoding judgement* $b \downarrow_{\texttt{Body}} e_{\text{body}}$. An inverse mapping can also be defined by the *body decoding judgement* $e_{\text{body}} \uparrow_{\texttt{Body}} b$. Rather than defining

[6] In Sec. 6, we add the machinery necessary for parse functions that are neither closed nor yet expanded.

Body explicitly, and these judgements inductively against that definition (which would be tedious and uninteresting), it suffices to take as a condition that there is an isomorphism between literal bodies and values of type Body mediated by these judgements (see supplement.)

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

The type abbreviated PrExpr classifies encodings of *proto-expressions*, $\grave{e}$ (pronounced "grave $e$".) The syntax of proto-expressions, defined in Fig. 11, will be described when we describe proto-expansion validation in Sec. 4.6. The mapping from proto-expressions to values of type PrExpr is defined by the *proto-expression encoding judgement*, $\grave{e} \downarrow_{\texttt{PrExpr}} e$. An inverse mapping is defined by the *proto-expression decoding judgement*, $e \uparrow_{\texttt{PrExpr}} \grave{e}$. Again, rather than picking a particular definition of PrExpr and defining the judgements above inductively against it, we 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 semantically 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, $\tau'$, 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 $\Psi$ is an *expression TLM definition context*.

A TLM identifier expansion context, $\mathcal{A}$, is a finite function mapping each TLM identifier $\hat{a} \in \mathrm{dom}(\mathcal{A})$ to the *TLM identifier expansion*, $\hat{a} \rightsquigarrow a$, for some *TLM name*, $a$. 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, $\Psi$, is a finite function mapping each TLM name $a \in \mathrm{dom}(\Psi)$ to an *expanded seTLM definition*, $a \hookrightarrow \texttt{setlm}(\tau; e_{\mathrm{parse}})$, where $\tau$ is the seTLM's type annotation, and $e_{\mathrm{parse}}$ is its parse function. We define $\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{setlm}(\tau; e_{\mathrm{parse}})$, when $\hat{\Psi} = \langle \mathcal{A}; \Psi \rangle$, as an abbreviation of $\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \texttt{setlm}(\tau; e_{\mathrm{parse}}) \rangle$.

The spTLM definition form operates analagously. The rule is reproduced below, and the details, which mirror those for seTLMs, are in the supplement. Notice that the spTLM context, $\hat{\Phi}$, rather than the $\hat{\Psi}$ is updated. This allows expression and pattern TLMs to use the same identifiers.

EE-DEF-SPTSM

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \text{ type} \qquad \emptyset\,\emptyset \vdash e_{\mathrm{parse}} : \texttt{parr(Body;ParseResultSP)} \qquad e_{\mathrm{parse}} \Downarrow e'_{\mathrm{parse}} \qquad \hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi},\hat{a}\rightsquigarrow a \hookrightarrow \texttt{sptlm}(\tau; e'_{\mathrm{parse}})} \hat{e} \rightsquigarrow e : \tau'}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for patterns by static}\ e_{\mathrm{parse}}\ \texttt{in}\ \hat{e} \rightsquigarrow e : \tau'}$$

## 4.5 TLM Application

The unexpanded expression form for applying an seTLM named $\hat{a}$ to a literal form with literal body $b$ is $\hat{a}\ \texttt{[|}b\texttt{|]}$. The typed expansion rule governing seTLM application is below:

EE-AP-SETSM

$$\frac{b \downarrow_{\texttt{Body}} e_{\mathrm{body}} \qquad e_{\mathrm{parse}}(e_{\mathrm{body}}) \Downarrow \texttt{inj[SuccessE]}(e_{\mathrm{proto}}) \qquad e_{\mathrm{proto}} \uparrow_{\texttt{PrExpr}} \grave{e} \qquad \mathrm{seg}(\grave{e})\ \text{segments}\ b \qquad \emptyset\,\emptyset \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e : \tau}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi}', \hat{a}\rightsquigarrow a \hookrightarrow \texttt{setlm}(\tau; e_{\mathrm{parse}});\hat{\Phi}} \hat{a}\ \texttt{[|}b\texttt{|]} \rightsquigarrow e : \tau}$$

The first premise determines the encoding of the literal body, $e_{\mathrm{body}}$, which, as discussed above, is a value of type Body.

$$
\begin{array}{lrcl}
\text{PrTyp} & \grave{\tau} & ::= & t \mid \texttt{prparr}(\grave{\tau};\grave{\tau}) \mid \texttt{prall}(t.\grave{\tau}) \mid \texttt{prrec}(t.\grave{\tau}) \mid \texttt{prprod}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i\in L}) \\
& & \mid & \texttt{prsum}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i\in L}) \mid \texttt{splicedt}[m;n] \\
\text{PrExp} & \grave{e} & ::= & x \mid \texttt{prlam}\{\grave{\tau}\}(x.\grave{e}) \mid \texttt{prap}(\grave{e};\grave{e}) \mid \texttt{prtlam}(t.\grave{e}) \mid \texttt{prtap}\{\grave{\tau}\}(\grave{e}) \mid \texttt{prfold}(\grave{e}) \\
& & \mid & \texttt{prtpl}\{L\}(\{i \hookrightarrow \grave{e}_i\}_{i\in L}) \mid \texttt{prinj}[\ell](\grave{e}) \mid \texttt{prmatch}[n](\grave{e};\{\grave{r}_i\}_{1\le i\le n}) \mid \texttt{splicede}[m;n;\grave{\tau}] \\
\text{PrRule} & \grave{r} & ::= & \texttt{prrule}(p.\grave{e}) \\
\text{PrPat} & \grave{p} & ::= & \texttt{prwildp} \mid \texttt{prfoldp}(p) \mid \texttt{prtplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i\in L}) \mid \texttt{prinjp}[\ell](\grave{p}) \mid \texttt{splicedp}[m;n;\grave{\tau}]
\end{array}
$$

Fig. 11. Syntax of $\lambda^{\mathbf{S}}_{\mathbf{TLM}}$ proto-expansions. Proto-expansion terms are ABTs identified up to alpha-equivalence.

The second premise applies the parse function $e_{\text{parse}}$ to the encoding of the literal body. If parsing succeeds, i.e. a value of the form $\texttt{inj[SuccessE]}(e_{\text{proto}})$ results from evaluation, then $e_{\text{proto}}$ will be a value of type PrExpr (assuming a well-formed expression TLM context, by application of the Type Safety assumption.) We call $e_{\text{proto}}$ 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.

The fourth premise determines the segmentation of the proto-expansion, $\text{seg}(\grave{e})$, and ensures that it is valid with respect to $b$ via the predicate $\psi$ segments $b$, which checks that each segment in the finite set of segments $\psi$ has non-negative length and is within bounds of $b$, and that the segments in $\psi$ 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 spTLM application is analogous:

$$
\textsc{pe-ap-sptsm}\quad
\frac{\begin{array}{l} b \downarrow_{\text{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj[SuccessP]}(e_{\text{proto}}) \qquad e_{\text{proto}} \uparrow_{\text{PrPat}} \grave{p} \\ \text{seg}(\grave{p}) \text{ segments } b \qquad \grave{p} \rightsquigarrow p : \tau \dashv\!\!\shortparallel^{\hat{\Delta};\hat{\Phi};b} \hat{\Gamma} \end{array}}{\hat{\Delta} \vdash_{\hat{\Phi}', \hat{a} \rightsquigarrow a \hookrightarrow \texttt{sptlm}(\tau; e_{\text{parse}})} \hat{a}\,[\,|\,b\,|\,] \rightsquigarrow p : \tau \dashv\!\!\shortparallel \hat{\Gamma}}
$$

## 4.6 Proto-Expansion Validation

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

| | |
|---|---|
| $\Delta \vdash^{\mathbb{T}} \grave{\tau} \rightsquigarrow \tau$ type | $\grave{\tau}$ has well-formed expansion $\tau$ |
| $\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau$ | $\grave{e}$ has expansion $e$ of type $\tau$ |
| $\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r} \rightsquigarrow r : \tau \Longmapsto \tau'$ | $\grave{r}$ has expansion $r$ taking values of type $\tau$ to values of type $\tau'$ |
| $\grave{p} \rightsquigarrow p : \tau \dashv\!\!\shortparallel^{\mathbb{P}} \hat{\Gamma}$ | $\grave{p}$ has expansion $p$ matching against $\tau$ generating assumptions $\hat{\Gamma}$ |

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. Rules EE-AP-SETSM and PE-AP-SPTSM) for when validation encounters spliced terms. *Type splicing scenes*, $\mathbb{T}$, are of the form $\hat{\Delta}$; $b$. *Expression splicing scenes*, $\mathbb{E}$, are of the form $\hat{\Delta}$; $\hat{\Gamma}$; $\hat{\Psi}$; $\hat{\Phi}$; $b$. *Pattern splicing scenes*, $\mathbb{P}$, are of the form $\hat{\Delta}$; $\hat{\Phi}$; $b$.

*4.6.1 Common Forms.* Most of the proto-expansion forms mirror corresponding expanded forms. The rules governing proto-expansion validation for these common forms correspondingly mirror the typing rules. They are given in the supplement. Splicing scenes 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.

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

$$\textsc{ptv-spliced}\quad \frac{\mathsf{parseUTyp}(\mathsf{subseq}(b;m;n)) = \hat{\tau} \qquad \langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle \vdash \hat{\tau} \rightsquigarrow \tau \ \mathsf{type} \qquad \Delta \cap \Delta_{\mathsf{app}} = \emptyset}{\Delta \vdash^{\langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle; b} \mathtt{splicedt}[m;n] \rightsquigarrow \tau \ \mathsf{type}}$$

This first premise of this rule parses out the requested segment of the literal body, $b$, to produce an unexpanded type, $\hat{\tau}$. It then invokes type expansion to ensure that this unexpanded type is well-formed *under the application site context*, $\langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle$, but *not* the expansion-local type formation context, $\Delta$. The final premise requires that the application site type formation context is 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. Collectively, this ensures that type variable capture does not occur (we will formally state this property in the next section.)

The rule for references to spliced unexpanded expressions is fundamentally analagous:

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

We first splice out the requested segment. The second premise expands the type annotation under an empty context, because the type annotation must be meaningful at the application site (so, independent of $\Delta$ and $\Gamma$) and not itself make any assumptions about the application site context. Spliced types can appear in the annotation. The third premise performs typed expansion of the spliced unexpanded expression under the application site contexts, but not the expansion-local contexts. The final two premises ensure that these contexts are disjoint, again to force capture avoidance.

The rule for references to spliced unexpanded patterns is entirely analagous:

$$\textsc{ppv-spliced}\quad \frac{\mathsf{parseUPat}(\mathsf{subseq}(b;m;n)) = \hat{p} \qquad \emptyset \vdash^{\hat{\Delta}; b} \grave{\tau} \rightsquigarrow \tau \ \mathsf{type} \qquad \hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\Vdash \hat{\Gamma}}{\mathtt{splicedp}[m;n;\grave{\tau}] \rightsquigarrow p : \tau \dashv\Vdash^{\hat{\Delta}; \hat{\Phi}; b} \hat{\Gamma}}$$

## 4.7 Metatheory

*4.7.1 Typed Expansion.* Let us now return to Theorem 4.1, the typed expression expansion theorem that was mentioned at the end of Sec. 4.3. 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 the three rules just described.)

Theorem 4.2 (Typed Expression Expansion (Strong)).

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

*(2) If* $\Delta \ \Gamma \vdash^{\langle \mathcal{D}; \Delta_{app} \rangle; \langle \mathcal{G}; \Gamma_{app} \rangle; \langle \mathcal{A}; \Psi \rangle; b} \grave{e} \rightsquigarrow e : \tau$ *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.) Such combined contexts can only be formed if the constituents 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. More specifically, the sum of the lengths of the literal bodies that appear in the term strictly decreases each time you perform a nested TLM application because some portion of the term has to be consumed by the TLM name and the delimiters. The details are given in the supplemental material. A similar argument is needed to prove Typed Pattern Expansion:

THEOREM 4.3 (TYPED PATTERN EXPANSION (STRONG)).

*(1) If* $\langle \mathcal{D}; \Delta \rangle \vdash_{\langle \mathcal{A}; \Phi \rangle} \hat{p} \rightsquigarrow p : \tau \dashv\Vdash \langle \mathcal{G}; \Gamma \rangle$ *then* $\Delta \vdash p : \tau \dashv\Vdash \Gamma$.

*(2) If* $\hat{p} \rightsquigarrow p : \tau \dashv\Vdash^{\langle \mathcal{D}; \Delta \rangle; \langle \mathcal{A}; \Phi \rangle; b} \langle \mathcal{G}; \Gamma \rangle$ *then* $\Delta \vdash p : \tau \dashv\Vdash \Gamma$.

*4.7.2 seTLM Reasoning Principles.* The following theorem summarizes the abstract reasoning principles that programmers can rely on when applying an seTLM. Informal descriptions of the labeled clauses are given inline, in gray boxes.

THEOREM 4.4 (SETLM REASONING PRINCIPLES). *If* $\langle \mathcal{D}; \Delta \rangle \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}} \hat{a}\ [\,|\,b\,|\,] \rightsquigarrow e : \tau$ *then:*

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

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

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

*(3)* $e_{parse}(e_{body}) \Downarrow \texttt{inj[SuccessE]}(e_{proto})$

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

*(7)* ***(Typing 2)*** $\{\langle \mathcal{D}; \Delta \rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m_i'; n_i')) \rightsquigarrow \tau_i'\ \mathsf{type}\}_{0 \le i < n_{ty}}$ *and* $\{\Delta \vdash \tau_i'\ \mathsf{type}\}_{0 \le i < n_{ty}}$

*Each spliced type has a well-formed expansion.*

*(8)* ***(Typing 3)*** $\{\emptyset \vdash^{\langle \mathcal{D}; \Delta \rangle; b} \grave{\tau}_i \rightsquigarrow \tau_i\ \mathsf{type}\}_{0 \le i < n_{exp}}$ *and* $\{\Delta \vdash \tau_i\ \mathsf{type}\}_{0 \le i < n_{exp}}$

*Each type annotation on a reference to a spliced expression has a well-formed expansion.*

*(9)* ***(Typing 4)*** $\{\langle \mathcal{D}; \Delta \rangle \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}} \mathsf{parseUExp}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow e_i : \tau_i\}_{0 \le i < n_{exp}}$ *and* $\{\Delta\ \Gamma \vdash e_i : \tau_i\}_{0 \le i < n_{exp}}$

*Each spliced expression has a well-typed expansion consistent with the type annotation in the segmentation.*

*(10)* ***(Capture Avoidance)*** $e = [\{\tau_i'/t_i\}_{0 \le i < n_{ty}}, \{e_i/x_i\}_{0 \le i < n_{exp}}]e'$ *for some variables* $\{t_i\}_{0 \le i < n_{ty}}$ *and* $\{x_i\}_{0 \le i < n_{exp}}$, *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)*** $\mathsf{fv}(e') \subset \{t_i\}_{0 \le i < n_{ty}} \cup \{x_i\}_{0 \le i < n_{exp}}$

*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.*

The proof, which involves auxiliary lemmas about the decomposition of proto-types and proto-expressions, is given in the supplement.

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 is the case in the typical formal accounts of hygiene (Adams 2015; Clinger and Rees 1991; Dybvig et al. 1992; Herman 2010; Herman and Wand 2008; Kohlbecker et al. 1986). Instead, we used standard notions of capture avoiding substitution and free variables combined with the context disjointness conditions in the rules above. This is possible only because we keep track of spliced terms explicitly in the proto-expansion, rather than going straight to the final expansion. In fact, doing so is critical for TLMs – there is no notion of alpha-conversion for partially parsed terms, so any notion of hygiene that relies on this notion would be inapplicable.

*4.7.3 spTLM Reasoning Principles.* Finally, the following theorem summarizes the abstract reasoning principles available to programmers when applying an spTLM. Most of the labeled clauses are analagous to those described above, so we omit their descriptions.

Theorem 4.5 (spTLM Reasoning Principles). *If* $\hat{\Delta} \vdash_{\hat{\Phi}} \hat{a}\ [\,|\,b\,|\,] \rightsquigarrow p : \tau \dashv\Vdash \hat{\Gamma}$ *where* $\hat{\Delta} = \langle \mathcal{D}; \Delta \rangle$ *and* $\hat{\Gamma} = \langle \mathcal{G}; \Gamma \rangle$ *then all of the following hold:*

*(1)* ***(Typing 1)*** $\hat{\Phi} = \hat{\Phi}', \hat{a} \rightsquigarrow a \hookrightarrow \texttt{sptlm}(\tau; e_{parse})$ *and* $\Delta \vdash p : \tau \dashv\Vdash \Gamma$

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

*(3)* $e_{parse}(e_{body}) \Downarrow \texttt{inj[SuccessP]}(e_{proto})$

*(4)* $e_{proto} \uparrow_{\text{PrPat}} \dot{p}$

*(5)* ***(Segmentation)*** $\text{seg}(\dot{p})$ segments $b$

*(6)* $\text{seg}(\dot{p}) = \{\texttt{splicedt}[n'_i; m'_i]\}_{0 \le i < n_{ty}} \cup \{\texttt{splicedp}[m_i; n_i; \dot{\tau}_i]\}_{0 \le i < n_{pat}}$

*(7)* ***(Typing 2)*** $\{\hat{\Delta} \vdash \text{parseUTyp}(\text{subseq}(b; m'_i; n'_i)) \rightsquigarrow \tau'_i\ \text{type}\}_{0 \le i < n_{ty}}$ *and* $\{\Delta \vdash \tau'_i\ \text{type}\}_{0 \le i < n_{ty}}$

*(8)* ***(Typing 3)*** $\{\emptyset \vdash^{\hat{\Delta}; b} \dot{\tau}_i \rightsquigarrow \tau_i\ \text{type}\}_{0 \le i < n_{pat}}$ *and* $\{\Delta \vdash \tau_i\ \text{type}\}_{0 \le i < n_{pat}}$

*(9)* ***(Typing 4)*** $\{\hat{\Delta} \vdash_{\hat{\Phi}} \text{parseUPat}(\text{subseq}(b; m_i; n_i)) \rightsquigarrow p_i : \tau_i \dashv\Vdash \langle \mathcal{G}_i; \Gamma_i \rangle\}_{0 \le i < n_{pat}}$ *and* $\{\Delta \vdash p_i : \tau_i \dashv\Vdash \Gamma_i\}_{0 \le i < n_{pat}}$

*(10)* ***(Visibility)*** $\mathcal{G} = \biguplus_{0 \le i < n_{pat}} \mathcal{G}_i$ *and* $\Gamma = \bigcup_{0 \le i < n_{pat}} \Gamma_i$

*The hypotheses generated by the TLM application are exactly those generated by the spliced patterns.*

## 4.8 Retrospective

Let us step back and briefly review what we have accomplished so far. First, we defined an entirely standard expanded language (Sec. 4.1). Then we mirrored the forms in this language to form the common forms of the unexpanded language (Sec. 4.2), taking care to handle identifiers carefully (Sec. 4.3). We also added forms for defining TLMs (Sec. 4.4). We then considered TLM application, which invokes a parse function to programmatically produce an encoding of a proto-expansion (Sec. 4.5). Each proto-expansion is then decoded, validated and inductively expanded (Sec. 4.6) to establish the abstract reasoning principles just described (Sec. 4.7).

# 5 Parametric TLMs (pTLMs)

Simple TLMs operate at a single specified type and the proto-expansions they generate must be closed (cf. the final premise of Rule ee-ap-setsm in the Sec. 4.5.) While this suited our purposes in the previous section because we intend $\lambda^{\mathbf{S}}_{\mathbf{TLM}}$ to communicate the essential concepts, it is not realistic for an ML-like language to impose these limitations. This section introduces *parametric TLMs*. Parametric TLMs can be defined over a type- and module-parameterized family of types, and the proto-expansions they generate can refer to supplied type and module parameters.

### 5.1 Parametric TLMs By Example

Consider the following Reason module type (a.k.a. signature), which specifies an abstract data type (Harper 1997; Liskov and Zilles 1974) of string-keyed polymorphic immutable dictionaries:

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

Let us now define a TLM that is parametric over all implementations of this signature, `D : DICT`, and also over all choices of the range type, `'a`:

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

Assuming some module that implements this signature, `HashDict : DICT`, and some values `v1 : int` and `v2 : int`, we can apply `$dict` as follows:

```
$dict HashDict int [| "key1" => v1; "key2" => v2 |]
```

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 terms (Schwerdfeger and Van Wyk 2009), so there can never be an ambiguity about where a key expression ends.

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`, as follows:

```
syntax $dict' = $dict HashDict;
$dict' int [| "key1" => v1; "key2" => v2 |];
```

The proto-expansion generated for the TLM application above might be:

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

The generated proto-expansion must truly be parametric, i.e. it must be valid for all modules `D : DICT` and types `'a`. It is only after proto-expansion validation that we substitute the actual parameters, here `HashDict` for `D` and `int` for `'a`, into the final expansion. Additionally, substitution occurs on the type annotations on references to spliced terms before recursively expanding those terms (otherwise, the expressions `v1` and `v2` above would need to be well-typed for all types, `'a`.)

Module parameters are also useful for giving the expansion access to helper functions in a context-independent manner. For example, in Sec. 2.3.2 we discussed the problem of applying datatype constructors like `H1Element`, because they are introduced into the context as variables in most ML-like languages. With module parameters, it is possible to sidestep this problem by passing in a module containing the datatype constructors. Because this will be common in practice, we provide the shorthand `syntax $a at T using t1, (*...*), tn, X1, (*...*), Xm by static { ... }` for the parameterization of `$a` by $n$ type variable parameters and $m$ module variable parameters, followed by the partial application of the given parameters. Additionally, for the type variable parameters that refer to datatypes, the constructors are bound automatically in the expansion. This explicit form of capture is reminiscent of work on explicit capture for distributed functions by Miller et al. (2014). Although a bit cumbersome, asking TLM providers to make their intent explicit allows them to avoid having to reason about the computations that the parse function performs on

syntax tree representations when refactoring code. Otherwise, proto-expansion representations would need to be built primitively into the binding structure of the language. We want to leave the semantics of OCaml alone when incorporating TLMs into Reason, so this would not be feasible.

It is important to note that module parameters are accessible by expansions, but not by parse functions directly, because the applied module parameters will not have been dynamically instantiated when typed expansion occurs. We will discuss a distinct mechanism for providing helper functions to parse functions in Sec. 6.

### 5.2 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. 4, 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, $\sigma$; 2) constructions, $c$, classified by kinds, $\kappa$; and 3) expressions classified by types, which are constructions of kind $\mathtt{Type}$ (we use metavariables $\tau$ 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 (2012), which in turn are based on early work by MacQueen (1984, 1986), subsequent work on the phase splitting interpretation of modules (Harper et al. 1989) and on using dependent singleton kinds to track type identity (Crary 2009; Stone and Harper 2006), and finally on formal developments by Dreyer (2005) and Lee et al. (2007). 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. 12.

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

Fig. 12. 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 $\sigma$. 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.

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

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, $\epsilon$, 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*, $\Omega_{\text{app}} \vdash^{\text{Exp}}_{\Psi} \epsilon \Downarrow \epsilon_{\text{normal}}$. This is defined in terms of a structural operational semantics (Plotkin 2004) with two stepping rules:

$$\textsc{eps-dyn-apmod-subst-e}\quad \frac{}{\Omega \vdash^{\text{Exp}}_{\Psi} \texttt{apmod}\{X\}(\texttt{absmod}\{\sigma\}(X'.\epsilon)) \mapsto [X/X']\epsilon} \qquad \textsc{eps-dyn-apmod-steps-e}\quad \frac{\Omega \vdash^{\text{Exp}}_{\Psi} \epsilon \mapsto \epsilon'}{\Omega \vdash^{\text{Exp}}_{\Psi} \texttt{apmod}\{X\}(\epsilon) \mapsto \texttt{apmod}\{X\}(\epsilon')}$$

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, $\Psi$.

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.

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

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, $\tau_{\text{proto}}$, that uses the same free variables to stand for the parameters; 3) a *substitution*, $\omega$, that pairs the applied parameters from $\epsilon_{\text{normal}}$ with the corresponding variables generated when peeling away the binders in $\dot{e}$; and 4) a corresponding *parameter context*, $\Omega_{\text{params}}$, that tracks the signatures of these variables. The two rules governing the proto-expression deparameterization judgement are below:

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

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

This judgement can be pronounced "when applying peTLM $\epsilon$, $\dot{e}$ has deparameterization $\grave{e}$ leaving $\rho$ with parameter substitution $\omega$". 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 $\Omega_{\text{app}}$, 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, $\Omega_{\text{param}}$ (rather than the empty context, as was the case in $\lambda^{\mathbf{S}}_{\mathbf{TLM}}$.) The conclusion of the rule applies the parameter substitution, $\omega$, 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. 5.1, these annotations might refer to the parameters, so the parameter substitution, $\omega$, 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:

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

(This is only sensible because we maintain the invariant that $\Omega$ is always an extension of $\Omega_{\mathsf{params}}$.)

The calculus enjoys metatheoretic properties analagous to those described in Sec. 4.7, 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 4.4, 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 5.1 (peTLM Reasoning Principles). *If* $\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}\ \mathtt{[|}b\mathtt{|]} \rightsquigarrow e : \tau$ *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)*** $\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{e} \rightsquigarrow \epsilon \ @\ \mathtt{type}(\tau)$ *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)* $e_{pproto} \uparrow_{\mathsf{PPrExpr}} \dot{e}$
*(10)* $\Omega_{app} \vdash^{\mathsf{Exp}}_{\Psi} \dot{e} \hookrightarrow_{\epsilon_{normal}} \grave{e}\ ?\ \mathtt{type}(\tau_{proto}) \dashv \omega : \Omega_{params}$
*(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)*** $\mathsf{fv}(e'') \subset \{k_i\}_{0\le i<n_{kind}} \cup \{u_i\}_{0\le i<n_{con}} \cup \{x_i\}_{0\le i<n_{exp}} \cup dom(\Omega_{params})$

## 6 Static Evaluation

There is one more major impracticality that we can now address. So far, we have assumed that parse functions are closed expanded expressions. This assumption simplifies matters formally, but

in practice it is quite difficult, because this means that TLM definitions cannot themselves access any libraries nor use TLMs internally. To address this problem, we introduce a *static environment*.

### 6.1 The Static Environment

Fig. 13 shows an example of a module, `ParserCombos`, that defines a number of parser combinators following Hutton (1992). The **`static`** qualifier indicates that this module is bound for use only within similarly qualified values, including in particular parse functions of subsequent TLM definitions.

```
static module ParserCombos = {
  type parser('c, 't) = list('c) -> list('t, list('c));
  let alt : parser('c, 't) -> parser('c, 't) -> parser('c, 't);
  /* ... */
};
syntax $a at t by static { fun(b) => ParserCombos.alt /* OK ... */ };
static val y = ParserCombos.alt /* OK ... */;
val z = /* ... neither ParserCombos nor y are available here */;
```

Fig. 13. Binding static modules and values for use within parse functions.

The values that arise when the static phase runs do not persist from "compile-time" to "run-time", so we do not need a full staged computation system, e.g. as described by Taha (1999). Instead, a sequence of static bindings operates like a read-evaluate-print loop (REPL) scoped according to the program structure, in that each static expression is evaluated immediately and the evaluated values are tracked by a *static environment*. The static environment is discarded after typed expansion.

A language designer might choose to restrict the external effects available to static terms in some way, e.g. to ensure deterministic builds. It might also be helpful to restrict mutable state shared between TLMs to prevent undesirable TLM application order dependencies. On the other hand, these features, if used for the purposes of caching, might speed up typed expansion. These are orthogonal design decisions.

### 6.2 Applying TLMs Within TLM Definitions

TLMs and TLM abbreviations can also be qualified with the **`static`** keyword, which marks them for use within subsequent static expressions and patterns. Let us consider some examples of relevance to TLM providers.

*6.2.1 Quasiquotation.* TLMs must construct values of type `proto_expr` or `proto_pat`. Constructing values of these types explicitly can have high syntactic cost. To decrease this cost, we can define TLMs that provide support for *quasiquotation syntax* similar to that built in to languages like Lisp (Bawden 1999) and Scala (Shabalin et al. 2013). The following TLM defines quasi-quotation for encodings of proto-expressions:

```
static syntax $proto_expr at proto_expr by static { /* ... */ };
```

For example, `$proto_expr \`g x ^x\`` might have expansion `App(App(Var "g", Var "x"), x)`. Notice that prefixing a variable (or parenthesized expression) with `^` serves to splice in its value, which here must be of type `proto_expr` (though in other syntactic positions, it might be `proto_typ`.) This is also known as *anti-quotation*.

*6.2.2 Parser Generators.* Abstractly, a grammar-based parser generator is a module matching the signature `PARSEGEN` defined below:

```
module type PARSEGEN = {
  type grammar('a);
  /* ... operations on grammars ... */
  val generate : grammar('a) -> (body -> parse_result('a));
};
```

Rather than constructing a grammar equipped with semantic actions using the associated operations (whose specifications are elided in `PARSEGEN`), we wish to use a syntax for context-free grammars that follows standard conventions. We can do so by defining a static parametric TLM:

```
static syntax $grammar(P : PARSEGEN)(type 'a) at P.grammar('a) /*...*/
```

To support splicing, we need non-terminals that recognize unexpanded terms and produce the corresponding splice references, rather than the AST itself. This requires that the parser generator keep track of location information (as most production-grade parser generators already do for error reporting.) A more detailed example of this mechanism being used to define a TLM for a modular encoding of regular expressions is given in the supplement.

A grammar containing such non-terminals can serve as a *summary specification* – a human can simply be take this grammar as a specification of what the segmentation will be for every recognized string, rather than relying on an editor to communicate this information. The associated semantic actions can be held abstract as long as the system performs a simple check to ensure that the proto-expansion does mention each spliced expression from the corresponding production.

### 6.3 Static Evaluation, Formally

It is not difficult to extend $\lambda^{\mathbf{P}}_{\mathbf{TLM}}$ to account for static evaluation. Static environments, $\Sigma$, take the form $\omega : \hat{\Omega}; \hat{\Psi}; \hat{\Phi}$, where $\omega$ is a substitution. Each binding form is annotated with a *phase*, $\phi$, either `static` or `standard`. The rules for binding forms annotated with `standard` are essentially unchanged, differing only in that $\Sigma$ passes through opaquely. The rules for binding forms annotated with `static` are based on the corresponding `standard` phase rules, differing only in that 1) they operate on $\Sigma$ and 2) evaluation occurs immediately. Finally, the forms for TLM definition are modified so that the parse function is now an unexpanded, rather than an expanded expression. The substitution $\omega$ is applied to the parse function after it is expanded. The full details are defined as a small patch of $\lambda^{\mathbf{P}}_{\mathbf{TLM}}$ called $\lambda^{\mathbf{PH}}_{\mathbf{TLM}}$ in the supplement.

### 6.4 Library Management

In the examples above, and in our formal treatment, we explicitly qualified various definitions with the **`static`** keyword to make them available within static values. In practice, we would like to be able to use libraries within both static values and standard values as needed without duplicating code. This can be accomplished either by the package manager (e.g. SML/NJ's CM (Blume 2002), extended with phase annotations) or by allowing one to explicitly lower an instance of a module define in the standard phase for use also in the static phase, as in a recent proposal for modular staging macros in OCaml (Yallop and White 2015).

TLMs definitions can be exported from the top level of packages, but they cannot be exported from within ML-style modules because that would require that they also appear in signatures, and that, in turn, would complicate reasoning about signature equivalence, since TLM definitions contain arbitrary parse functions. That said, it should be possible to export TLM *abbreviations* from modules, since they refer to TLM definitions only through symbolic names. We have not yet formalized this intuition, but the work of Culpepper et al. (2005, 2007) considered a closely related question: how should Typed Scheme's macros interact with its unit (i.e. package) system.

## 7 Related Work

### 7.1 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. For example, Ur/Web's syntax (Fig. 1) is a library-specific dialect of Ur's syntax (Chlipala 2010, 2015).

There are hundreds of syntax definition systems of various design. Notable examples include grammar-oriented systems like Camlp4 (Leroy et al. 2014), Copper (Schwerdfeger and Van Wyk 2009; Van Wyk and Schwerdfeger 2007), SugarJ/Sugar*/SoundExt (Erdweg et al. 2011; Erdweg and Rieger 2013; Lorenzen and Erdweg 2013, 2016), as well as various parser combinator systems (Hutton 1992). 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 (Flatt 2012), or the import mechanism of SugarJ/Sugar*/SoundExt.)

The first problem with using a syntax definition system to directly extend the context-free syntax of a language is that clients cannot always deterministically combine the resulting extended syntax dialects when they want to use the new forms that they define together, i.e. there can be syntactic conflicts. This is a significant problem because large programs use many separately developed libraries (Lämmel et al. 2011) and often do not fall cleanly into a single "problem domain".

One possible solution is found in Copper, which integrates a modular grammar analysis that guarantees that determinism is conserved when syntax dialects of a certain restricted class are combined Schwerdfeger and Van Wyk (2009). The caveat is that the constituent dialects 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 sidestep these complexities entirely because the context-free syntax of the language is fixed, and composition is via splicing rather than direct combination. The TLM provider can therefore modularly establish that the syntax definition that the TLM implements (possibly using a syntax definition system internally, as discussed in Sec. 6) is deterministic. There is no need for verbose marking tokens, and programmers can define scoped TLM abbreviations as discussed in Sec. 5.

Even putting aside the problem of syntactic conflict, there are questions about just how reasonable sprinkling many possibly unfamiliar library-specific literal forms throughout a program may be. The intuition was given by example in Sec. 1. To reiterate, more generally:

(1) **Responsibility**: Clients using a combined syntax dialect cannot easily determine which constituent extension is responsible for a given form, whereas TLM clients can follow the binding structure of the language in the usual manner.
(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) **Binding**: Clients of a context-free syntax dialect cannot be sure that the desugaring is context-independent and that spliced terms are capture avoiding, as TLM clients can.
(4) **Typing**: Clients of a syntax dialect cannot reason abstractly about types. 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.

The work of Lorenzen and Erdweg (2013, 2016) 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.

While the client can indirectly reason about binding (but not segmentation) by inspecting the derived typing rules, this is weaker than the strict hygiene and segmentation discipline that TLMs enforce. Another distinction is that TLMs do not support type-dependent expansions. The trade-off is that this allows expansions, and therefore segmentations, to be generated even when the program is ill-typed, as long as the static values are well-typed (this notion of *untyped expansion* was not considered formally above, but it follows straightforwardly from the fact that all of the premises to the TLM application rules except the final proto-expansion validation premises are independent of the typing context.) 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 and heavyweight theorem proving machinery. Finally, it would be difficult to define syntax extensions in SoundExt that operate over module-parameterized families of types. Parameters can be approximated using splicing, but there is no clear notion of "partial application".

### 7.2 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.

For example, OCaml's textual syntax now includes *preprocessor extension (ppx) points* used to identify terms that some term rewriting preprocessor must rewrite (Leroy et al. 2014). We could mark a string literal containing Ur/Web-style HTML syntax with a ppx annotation, `xml`, as follows:

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

There are problems reasoning modularly and abstractly about such constructions. More than one applied preprocessor might recognize this annotation (there are, in practice, many XML/HTML libraries), so the problems of **Responsibility** come 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 (Hart 1963), Template Haskell (Sheard and Peyton Jones 2002) and GHC's quasiquotation system (Mainland 2007) (which is based on Template Haskell), do not allow clients to reason abstractly about the remaining issues, again because the expansion that they produce is unconstrained. (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 (Herman 2010; Herman and Wand 2008), **Capture**, and they enforce application-site **Context Independence** (Adams 2015; Clinger and Rees 1991; Dybvig et al. 1992; Kohlbecker et al. 1986). However, this turns out to make 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! "<h1>Hello, {[first_name]}!</h1>")
```

The expansion of this macro will fail the check for context independence because `first_name` will appear as a free variable, no 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 expression of type **`string`**.

Much of the research on macro systems has been for languages in the LISP tradition (McCarthy 1978) that do not have rich static type structure. The formal macro calculus studied by Herman and Wand (2008) uses types to encode the binding structure of the generated expansion (in particular, to indicate which identifiers passed as arguments can be captured by which other arguments), but the expanded language itself does not have rich type structure (nor is this calculus capable of expressing new literal forms, for the reasons already discussed.) Research on typed *staging macro systems* like MetaML (Sheard 1999) and MacroML (Ganz et al. 2001) 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 useful mainly for performance reasons.) The Scala macro system is a hygienic macro system which supports "black-box" term rewriting macros. These 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. In any case, black-box macros also cannot be used to repurpose string literal forms because they are hygienic.

Some languages, including Scala (Odersky et al. 2008), build in *string splicing* (a.k.a. *string interpolation*) forms, or similar but more general *fragmentary quotation forms* (Slind 1991), e.g. the SML/NJ dialect of ML. 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 XHTML example, we have splicing at two different types using two different delimiters. These forms are also not available in pattern position.

This brings us to the most closely related work, that of Omar et al. (2014) on *type-specific languages* (TSLs). Like simple expression TLMs (Sec. 2), TSLs allow library providers to programmatically control the parsing of expressions of generalized literal form. 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 be defined at the same type without conflict, and can operate at any type, not just at nominal (i.e. abstract) types. In a subsequent short paper, Omar et al. (2015) proposed explicit application of simple expression TLMs (there called *typed syntax macros*) in a bidirectional typed setting (Pierce and Turner 2000), but this short proposal did not give any formal details. With TLMs, it is not necessary for the language to be bidirectionally typed – 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.) It should be possible to implicitly invoke TLMs based on the expected type in situations where the inference engine had enough information, but we leave this as an interesting direction for future work.

Another critical distinction is that the metatheory presented by Omar et al. (2014) establishes only that generated expansions are of the expected type (i.e. a variant of the Typed Expression Expansion theorem from Sec. 4.7.) It does not establish the remaining abstract reasoning principles that have been a major focus of this paper. In particular, there is no formal hygiene theorem (though it is discussed informally), and the mechanism does not guarantee that a valid segmentation will exist, nor associate types with segments. Finally, this prior work did not consider pattern matching, type functions, ML-style modules, parameters or static evaluation. All of these are important for integration into an ML- or Scala-like language, and the focus of most of this paper.

## 8 Discussion

The importance of specialized notation as a "tool for thought" has long been recognized (Iverson 1980). According to Whitehead, a good notation "relieves the brain of unnecessary work" and

"sets it free to concentrate on more advanced problems" (Cajori 1928), and indeed, advances in mathematics, science and programming have often been accompanied by new notation.

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

Previously, these two relief mechanisms were in tension – mechanisms that allowed programmers to express new notation would obscure the type and binding structure of the program text. TLMs resolve this tension for the broad class of literal forms that generalized literal forms subsume. This class includes all of the examples enumerated in Sec. 1 (up to the choice of outermost delimiter), the additional examples described in Sec. 6, the more detailed example in the supplement, and the many examples in the prior papers by Omar et al. (2014, 2015).

Of course, not all possible literal notation will prove to be in good taste, and we make no empirical claims about particular notation in this paper. TLMs leave the burden of establishing the value of any particular literal notation to individual library providers, rather than to the language designer. The reasoning principles that TLMs provide, which are the specific contributions of this paper, allow clients to "reason around" poor literal designs, using principles analagous to those already familiar to programmers in languages like these. The technical content of this paper, which formally establishes these reasoning principles, constitutes the first detailed type-theoretic account of a typed, hygienic macro system for an ML-like language, i.e. one with a rich static type system, support for pattern matching and an ML-like module system with type functions and abstract type members. Our own language integration efforts have been focused on an emerging alternative front-end for OCaml called Reason, but Scala supports analagous features (Amin et al. 2014) and would also be a suitable host for the TLM mechanism. The intended audience for this paper is language designers seeking a reasonable solution to the problem of user-defined literal notation.

There are several interesting avenues for future work beyond those already discussed. 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 term representations – has been studied in a variety of settings, e.g. in MetaML (Sheard 1999) and in the modal logic tradition (Davies and Pfenning 1996). We leave integration of these approaches into the TLM mechanism as future work. Our initial efforts aim to leave the semantics of OCaml unchanged, consistent with the philosophy of the Reason project.

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. (2017) on tracking bindings "backwards" from an expansion to the source program is likely relevant.

At several points in the paper, we allude to editor integration. However, several important questions having to do with TLM-specific syntax highlighting, incremental parsing and error recovery (Graham et al. 1979) remain to be considered. Another interesting direction would be to generalize TLMs to support non-textual notation in the setting of a structure editor. The semantic foundations for structure editors proposed recently by Omar et al. (2017) may guide such an effort.

Let us conclude by considering the oft-uttered phrase "syntax doesn't matter", which is generally taken to mean that syntactic concerns are orthogonal to various more important semantic concerns. Our hope is that the reader will leave this paper with an understanding that this is a rather simplistic proclamation. Instead, let us proclaim that "semantics matter for syntax too!"

## REFERENCES


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

Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of path-dependent types. In *OOPSLA*. ACM, 233–249.

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.

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

Matthias Blume. 2002. *The SML/NJ Compilation and Library Manager*. Available from http://www.smlnj.org/doc/CM/index.html.

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

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

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

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

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

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

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

Ryan Culpepper, Sam Tobin-Hochstadt, and Matthew Flatt. 2007. Advanced macrology and the implementation of Typed Scheme. In *Workshop on Scheme and Functional Programming*.

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

F. DeRemer and H. Kron. 1976. Programming-in-the-Large versus Programming-in-the-Small. *IEEE Transactions on Software Engineering* 2 (1976), 80–86.

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

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

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

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

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

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

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

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

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

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

Robert Harper. 2016. *Practical Foundations for Programming Languages* (2nd ed.). Cambridge University Press. https://www.cs.cmu.edu/~rwh/plbook/2nded.pdf

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

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

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

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

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

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

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

Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. 2011. Large-scale, AST-based API-usage analysis of open-source Java projects. In *ACM Symposium on Applied Computing (SAC)*. http://doi.acm.org/10.1145/1982185.1982471

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

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

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

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

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

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

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. DOI: http://dx.doi.org/10.1145/512644.512670

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

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

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

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. DOI: http://dx.doi.org/10.1007/978-3-662-44202-9_13

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

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

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

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

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

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

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

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

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

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

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

Dana Scott. 1980. Lambda calculus: some models, some philosophy. *Studies in Logic and the Foundations of Mathematics* 101 (1980), 223–265.

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

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

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

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

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

Walid Taha. 1999. *Multi-Stage Programming: Its Theory and Applications*. Ph.D. Dissertation. Oregon Graduate Institute of Science and Technology.

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

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

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. DOI: http://dx.doi.org/10.1145/376284.375783

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