Reasonably Programmable Syntax

ANONYMOUS AUTHOR(S)

When designing the surface syntax of a programming language, it is common practice to build in literal forms for a select few standard data structures, e.g. lists. The problem is that this practice neglects an unbounded assortment of other data structures for which literal forms might also be useful (consider, for example, sets, maps, matrices, regular expressions, HTML trees, URLs, SQL queries and chemical structures.) A less *ad hoc* design would be one that allows library providers to define new literal forms in a decentralized manner. However, the mechanisms available to programmers today, e.g. Camlp4 (Leroy et al. 2014) and Sugar* (Erdweg et al. 2011; Erdweg and Rieger 2013), are *unreasonable*. In particular, they do not support modular reasoning about syntactic determinism, so separately defined literal forms can and do conflict syntactically with one another. Moreover, clients cannot reason abstractly about types and binding when they encounter an unfamiliar literal form (like they can when they encounter an unfamiliar function being applied.) Instead, they must reason transparently, i.e. about the underlying expansion. This increases cognitive cost, defeating much of the purpose of convenient literal forms.

This paper formally details a more reasonable alternative: *typed literal macros* (TLMs). TLMs give library providers full programmatic control over both the parsing and expansion of expressions and patterns of *generalized literal form* (Omar et al. 2014) at a specified type or parameterized family of types. Expansion is strictly hygienic, meaning that 1) expansions cannot make any assumptions about the application site context; and 2) expansions do not reveal internal bindings to spliced terms or to the remainder of the program. This design ensures that clients are able to reason modularly about syntactic determinism and abstractly about the type and binding structure of the program. The system needs only to convey to clients (e.g. via secondary notation) the *segmentation* of each literal, which gives the locations of the spliced segments. We establish these abstract reasoning principles formally with a calculus of expressions, patterns and ML-style modules. This is the first detailed type-theoretic account of a typed, hygienic term-rewriting macro system for an ML-likelanguage.

or Scala-like?

1 Introduction

The surface syntax of a programming language serves as the user interface between human programmers and the language's semantics. As such, language designers often include various *literal forms* in the surface syntax that are designed to decrease the cognitive cost of constructing and pattern matching over values of selected types.

For example, in Standard ML (SML), the list expression literal `[x, y, z]` desugars to a term semantically equivalent 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] List pattern literals are analagous. Another example is Ur/Web, which extends Ur's textual syntax with expression and pattern literal forms for XHTML elements (Chlipala 2010, 2015). Figure 1 gives examles of XHTML literals containing "spliced" string expressions (Line 1) and XHTML expressions (Line 2). The desugarings of these literals, not shown, are substantially more verbose and obscure.

Third-party library providers are quite justifiably envious of the privileged status of semantically ordinary types that have been equipped with convenient literal forms by the language designer. After all, it is not difficult to find other examples of types for which programmers have invented specialized syntax. For example, 1) clients of a "collections" library might like not just list literals, but also vector, matrix, finite set and finite map literals following standard mathematical conventions; 2) clients of a "web programming" library in SML might like Ur/Web style XHTML literals; 3) Ur programmers might further like CSS literals, which Ur/Web does not provide; 4) a compiler writer might like "quotation" literals for the terms of the object language and various intermediate languages of interest; and 5) clients of a "chemistry" library might like chemical structure literals

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

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. XHTML literals with support for splicing at two different types are built primitively into Ur/Web (Chlipala 2015).

based on the SMILES standard (Anderson et al. 1987). Research by Bravenboer et al. (2007) suggests that literal forms for structured encodings of SQL queries and other query types can reduce the temptation to use string encodings ("stringly-typed programming") and therefore reduce the prevalence of string injection attacks, which are amongst the most common and serious security vulnerabilities on the web today (OWASP 2017). Omar et al. (2014) sampled strings from open source projects and found that at least 15% of them were parseable according to some readily apparent type-specific grammar (e.g. for SQL queries, regexes, file system paths, URLs and others.)

Given this profusion of possible literal forms, it is not sensible to ask the designers of general-purpose languages to keep up. Instead, there has been considerable interest in mechanisms that allow programmers to introduce new type-specific literal forms in a decentralized manner. Unfortunately, existing mechanisms ...

This paper addresses this established need by introducing a system of *typed literal macros (TLMs)* that allows programmers to define new literal syntax of nearly arbitrary form within an ML- or Scala-like language, i.e. a language with a rich static type system (Sec. 2), support for pattern matching (Sec. 3) and an ML-style module system or Scala-style object system with support for type members and type abstraction (Sec. 5). The examples in this paper are written in VerseML, a new dialect of ML that features TLMs.

Compared to existing approaches, TLMs are more *reasonable*. We justify this claim by (1) developing a collection of relevant reasoning principles based on the reasoning principles already available to client programmers when using an ML-like or Scala-like language without such a mechanism; (2) formally operationalizing these reasoning principles as metatheorems about a series of type-theoretic *TLM calculi* of increasing sophistication (Sec. 4-5); and (3) evaluating existing approaches – including syntax definition systems like Camlp4 (Leroy et al. 2014), SugarJ/Sugar* (Erdweg et al. 2011; Erdweg and Rieger 2013) and Copper (Schwerdfeger 2010; Schwerdfeger and Wyk 2009), term rewriting macro systems as found in a variety of languages, and the system of type-specific languages (TSLs) and typed syntax macros (TSMs) described by Omar et al. (2014, 2015) – relative to these reasoning principles (Sec. 6). We conclude by discussing limitations and future research directions in Sec. 7. A complete set of definitions and associated proofs are available in the supplemental material. This is the first detailed type-theoretic accout of a typed, hygienic term-rewriting macro system for an ML- or Scala-like language.

er citation

refs

ne and justify claim

1.1 Contributions

This work introduces a system of **typed literal macros (TLMs)** that gives library providers substantially more syntactic control than existing typed term-rewriting macro systems while maintaining the ability to reason abstractly about types, binding and segmentation.

Our take on the running example is shown in Fig. 2: we apply a TLM named `$html` to *generalized literal forms* delimited by backticks. TLM names are prefixed by `$` to clearly them from variables. The semantics delegates control over the parsing and expansion of each literal body to the applied TLM during a semantic phase called *typed expansion*, which generalizes the usual typing phase.

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

Fig. 2. Two examples of a TLM being applied to a generalized literal form. We assume strings are primitive.

Generalized literal forms, which first arose in related work by Omar et al. (2014) that we will detail in Sec. 8, subsume a variety of common syntactic forms because the context-free syntax of the language only defines which outer delimiters are available (in this paper, we will mostly use backticks, but in general, there can be many choices.) *Literal bodies*, i.e. the characters between the delimiters, are otherwise syntactically unconstrained.

```
1  val w = compute_w ()
2  val x = compute_x ()
3  val y = $kquery `(!R)@&{&/x!/:2_!x}'!R}`
4
5
```

Fig. 3. TLMs make examples like the one from Fig. 11 more reasonable.

6

7 Choices regarding internal delimitation of spliced terms is not imposed by the language definition. Notice,
8 however, that Figure 2 reveals the locations of the spliced expressions by coloring them black. We have designed
9 our system so that a figure like this, which presents a *segmentation* of each literal body into spliced terms (in
10 black) and characters parsed in some other way by the applied TLM (in green)[2], can always be generated.
11 TLMs come equipped with useful principles of syntactic abstraction. We will more precisely characterize these
12 abstract reasoning principles as we proceed. For now, to develop some intuitions, consider Figure 3, which uses
13 TLMs to express the "unreasonable" example from Figure 11. Without examining the expansion of Line 3, we can
14 reason as follows:
15
16 (1) **(Responsibility)** The applied TLM, `$kquery`, is solely responsible for typed expansion of the literal body.
17 (2) **(Segmentation)** By examining the segmentation, we know that the two instances of `x` on Line 3 are
18 parsed as spliced expressions, whereas `R` is parsed in some other way peculiar to this form.
19 (3) **(Capture)** The system prevents capture, so the spliced expression `x` must refer to the binding of `x` on
20 Line 2 – it cannot capture an unseen binding introduced in the expansion of Line 3.
21 (4) **(Context Dependence)** The system enforces context independence, so the expansion of Line 3 cannot
22 rely on the fact that, for example, `w` is in scope.
23 (5) **(Typing)** An explicit type annotation on the definition of `$kquery` determines the type that every expan-
24 sion it generates will have. We will see an example of a TLM definition in the next section.
25 Moreover, each segment in the segmentation also comes paired with the type it is expected to have. This
26 information is usually not necessary to reason about typing, but it can be conveyed to the programmer
27 upon request by the program editor if desired.

28 The remainder of this document is organized as follows.

29 • Sec. 2 begins by going through the mechanics of defining a simple expression TLM and describes the
30 process of typed expansion and expansion validation by example.
31 • Sec. 3 does the same for a simple pattern TLM, i.e. a TLM that generates patterns that match values of a
32 specified type. The hygiene condition for pattern TLMs is interesting.
33 • Sec. 4 defines a calculus of simple expression and pattern TLMs and formally establishes the reasoning
34 principles implied above.
35 • Sec. 5 adds parameterized types and an ML-style module system and introduces *parametric TLMs* (pTLMs),
36 i.e. TLMs that take type and module parameters. Parameters serve two purposes:
37 (1) They allow for TLMs that operate not just at a single type, but over a type- and module-parameterized
38 family of types. For example, we consider a parametric TLM for dictionaries that operates uniformly
39 over different implementations of the dictionary signature, and also over different choices of the
40 co-domain type.
41 (2) Parameters also give expansions access to terms other than those explicitly spliced in to the literal
42 body. This is particularly convenient for working with abstract types *a la* ML. Partial parameter
43 application in TLM abbreviations lowers the cost of this explicit parameter-passing discipline.
44
45 [2]In general, spliced expressions might themselves apply TLMs, in which case the convention is to use a distinct color for unspliced segments
46 at each depth. For example, if strings were not primitive, we might write `` `<body>{heading ($str "World!")} ...</body>` ``.
47
48

This section culminates with a calculus that extends the calculus of Sec. 4 with ML-style modules and parametric TLMs.
- 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. 8 discusses additional related work, including closely related work by Omar et al. (2014, 2015) on type-specific languages. We conclude with a brief discussion of present limitations and future directions.

Due to space limitations, many of the standard or straightforward technical details of the calculi defined in Sec. 4 through Sec. 6 are relegated to the supplemental material. For convenience, the examples that we give are written in a full-scale functional language under development called VerseML. All examples written in VerseML should be understood to be informal motivating material for the subsequent formal material.

2 Simple Expression TLMs (seTLMs)

We will begin in this section by introducing *simple expression TLMs* (seTLMs), i.e. TLMs where expansions are expressions of a single specified type.

2.1 TLM Definitions

The definition of the seTLM named `$html` that was shown being applied in Fig. 2 takes the following form:

```
syntax $html at html by
  static fn(b : body) -> parse_result(proto_expr) =>
    (* Ur/Web-style HTML syntax parser here *)
in ... end
```

Every seTLM definition consists of a TLM name, here `$html`, a *type annotation*, here **`at`** `html`, and a *parse function* between **`by`** and **`in`**. The TLM definition is in scope between **`in`** and **`end`**. In practice, if an **`in`** clause is not provided, the definition is in scope until the end of the enclosing declaration (e.g. the enclosing function or module.) We will consider how TLM definitions are exported from libraries in Sec 6.

The parse function is a *static function* delegated responsibility over parsing the literal bodies of the literal forms to which the TLM is applied. Static functions, marked by the **`static`** keyword, are applied during the typed expansion process, so they cannot refer to the surrounding variable bindings (because those variables stand for dynamic values.) For now, we will simply assume that static functions are closed, and that they do not themselves make use of TLMs (we will eliminate these impractical limitations in Sec. 6.)

The parse function must have type `body -> parse_result(proto_expr)`. The input type, `body`, classifies encodings of literal bodies. In VerseML, literal bodies are sequences of characters, so it suffices to define `body` as an abbreviation for the `string` type, as shown in Figure 4. The return type is a labeled sum type, defined by applying the type function `parse_result` defined in Figure 4, that distinguishes between parse errors and successful parses. Let us consider these two possibilities in turn.

Parse Errors. If the parse function determines that the literal body is not well-formed (according to whatever syntax definition that it implements), it returns:

`ParseError {msg=`$e_{\texttt{msg}}$`, loc=`$e_{\texttt{loc}}$`}`

where $e_{\texttt{msg}}$ is an error message and $e_{\texttt{loc}}$ is a value of type `segment`, defined in Figure 4, that designates a segment of the literal body as the location of the error (van Deursen et al. 1993). This information is for use by VerseML compilers when reporting the error to the programmer (but it otherwise has no semantic significance.) We might alternatively have used an exception to signal a parse error, but sum types are easier to formalize.

Successful Parses. If parsing succeeds, the parse function returns

```
type body = string
type segment = {startIdx : nat, endIdx : nat} (* inclusive *)
type parse_result('a) = ParseError of { msg : string, loc : segment }
                      | Success of 'a
type var_t = string (* ... or some other encoding of variables ... *)
type proto_typ = TyVar of var_t
               | Arrow of proto_typ * proto_typ
               | (* ... *)
               | SplicedT of segment
type proto_expr = Var of var_t
                | Fn of var_t * proto_typ * proto_expr
                | Ap of proto_expr * proto_expr
                | (* ... *)
                | SplicedE of segment * proto_typ
```

Fig. 4. Definitions of various types used by TLM definitions. We assume that these type definitions are available ambiently.

`Success` e_{proto}

where e_{proto} is called the *encoding of the proto-expansion*. For expression TLMs, proto-expansions are *proto-expressions*, which are encoded as values of the recursive sum type `proto_expr` outlined in Figure 4. Most of the variants of `proto_expr` are individually uninteresting – they encode VerseML'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 Figure 4. More sophisticated encoding techniques for terms would cause no fundamental problems. It is only the `SplicedE` and `SplicedT` variants of these types that are interesting.

2.2 Splicing

When a parse function determines that a segment of the literal body should be taken as a spliced expression, it does not directly insert the syntax tree of that expression into the encoding of the expansion. Instead, the TLM must refer to the spliced expression *by its relative location* 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. 2, reproduced below:

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

Let us assume that `html` abbreviates a recursive labeled sum type having at least the cases

```
H1Element of list(html_attr) * list(html) | TextNode of string | ...
```

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

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

Here, **`spliced`**`<13; 22; string>` refers to the spliced expression `first_name` by location, rather than directly, and indicates that it must be of type `string`.

Requiring that TLMs refer to spliced expressions indirectly in this manner prevents them from "forging" spliced terms (i.e. claiming that an term is a spliced term when it does not in fact appear textually in the literal body.)

The *splice summary* of a proto-expression is the finite set of references to spliced terms contained within. For example, the summary of the proto-expression above is the finite set containing only `spliced<13; 22; string>`.

The *segmentation* of a proto-expression is the finite set of segments extracted from the splice summary, where a segment is simply a pair of natural numbers. For example, the segmentation of the proto-expansion above is $\{(13, 22)\}$. The semantics checks that all of the segments 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 the segmentation information to the programmer, e.g. by coloring non-spliced segments green as is our convention in this document. A program editor or pretty-printer can also communicate the type of each spliced term, as indicated in the splice summary, to the programmer upon request (for example, the Emacs and Vim packages for working with OCaml defer to the Merlin tool when the programmer requests the type of an expression.)

2.3 Proto-Expansion Validation

Three 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. The result of proto-expansion validation is the *final expansion*, which is simply the proto-expansion with references to spliced terms replaced with their corresponding final expansions.

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

```
let tmp = (* ... application site temporary ... *) in
$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 ... *) in
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, proto-expansion validation is capture-avoiding. In other words, it 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 ... *) in
let tmp' = (* ... expansion-internal temporary ... *) in
H1Element(tmp', f(tmp))
```

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

For TLM providers, the benefit of this mechanism is that they can name the variables used internally within expansions freely, without worrying about whether their chosen identifiers might shadow those that a client might have used at the application site. There is no need for a mechanism that generates "fresh variables".

TLM clients can, in turn, reliably reason about binding within every spliced expression without examining the expansion that the spliced expression appears within.

The trade-off is that this prevents library providers from intentionally introducing bindings into spliced terms. For example, Haskell's derived form 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 a command sequence. In VerseML, this cannot be expressed in the same way. Values can be communicated from the expansion to a spliced expression only via function arguments. We will return to this issue in Sec. 8.

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.

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

```
syntax $broken at rx by
  static fn(_) => Success (Var "x")
end
```

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, without considering the spliced terms.

One subtlety is that we assumed in the examples above that constructors like `H1Element` were available to the proto-expansion. In most 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 to ensure that the generated expansion is of the type specified in the TLM's type annotation;
(2) checking that the each spliced type is valid at the application site;
(3) checking that the type annotation on each spliced expression is valid at the application site; and
(4) checking each spliced expression against the provided 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:

```
match x with
| $html `<h1>{y}</h1>` -> y (* y : list(html) *)
| _ -> (* ... *)
end
```

Pattern TLM definitions look much like expression TLM definitions:

```
syntax $html at html for patterns by
  static fn(b : body) : parse_result(proto_pat) =>
    (* HTML pattern parser here *)
```

```
type proto_pat = (* no variable pattern form! *)
                 Wild
               | (* ... other standard pattern forms ... *)
               | SplicedP of segment * proto_typ
```

Fig. 5. Abbreviated definition of proto_pat.

```
  in (* ... *) end
```

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. Note 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*.[3] The return type of the parse function is `parse_result(proto_pat)`, rather than `parse_result(proto_expr)`. The type `proto_pat`, defined in part in Figure 5, classifies encodings of *proto-patterns*.

3.1 Splicing

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<5; 5; list(html)>)
```

3.2 Proto-Pattern Validation

Proto-pattern validation serves, like proto-expression validation, to maintain the ability to reason abstractly about binding and typing.

3.2.1 Binding. 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.

In our simple language, patterns have no way to refer to surrounding bindings (they only induce bindings). However, the 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 discussion.

Patterns cannot bind variables for use in sub-patterns, so we do not face the problem of capture.

3.2.2 Typing. 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 in the TLM type annotation;
(2) that each spliced pattern matches values of the type indicated in the splice summary; and
(3) that each of these types are themselves well-formed types at the application site.

4 Simple TLMs, Formally

Before continuing on to consider parametric TLMs, let us develop a calculus of simple expression and pattern TLMs called miniVerse$_S$. It is organized as an *unexpanded language*, or *UL*, defined by typed expansion to an

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

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

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

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

Fig. 7. Syntax of the miniVerse$_\mathbf{S}$ (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.

expanded language, or *XL*. Figures 6 and 7 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 miniVerse$_\mathbf{S}$ 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. The particularities of the XL are not critical to the ideas we will introduce below.

4.1.1 Statics of the Expanded Language. The *statics of the XL* is defined by hypothetical judgements of the following form:

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

Type formation contexts, Δ, are finite sets of hypotheses of the form $t\ \mathtt{type}$. *Typing contexts*, Γ, 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 τ. The judgements above are inductively defined in the supplemental material and validate standard lemmas, also given in the supplement: Weakening, Substitution and Decomposition.

4.1.2 Structural Dynamics. The *structural dynamics*, a.k.a. the *structural operational semantics* (Plotkin 2004), of miniVerse$_\mathbf{S}$ is defined as a transition system by judgements of the following form:

$e \mapsto e'$	e transitions to e'
$e\ \mathtt{val}$	e is a value

We also define auxiliary judgements for *iterated transition*, $e \mapsto^* e'$, and *evaluation* to a value, $e \Downarrow e'$. Additional auxiliary 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 for simplicity and again assume that standard lemmas, stated in the supplement, hold: Canonical Forms, Progress and Preservation.

4.2 Syntax of the Unexpanded Language

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

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 the existence of partial metafunctions $\mathsf{parseUTyp}(b)$, $\mathsf{parseUExp}(b)$ and $\mathsf{parseUPat}(b)$ that go from character sequences, b, to unexpanded types, expressions and patterns, respectively.

Condition 4.1 (Textual Representability).

(1) For each $\hat{\tau}$, there exists b such that $\mathsf{parseUTyp}(b) = \hat{\tau}$.
(2) For each $\hat{e}$, there exists b such that $\mathsf{parseUExp}(b) = \hat{e}$.
(3) For each $\hat{p}$, there exists b such that $\mathsf{parseUPat}(b) = \hat{p}$.

4.3 Typed Expansion

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

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

These judgements are inductively defined in the supplement. We will reproduce the interesting rules below.

Common Forms. Most of the unexpanded forms in Figure 6 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. The typed expansion rules that handle these common forms mirror the corresponding typing rules. For example, the rules for variables and lambdas are reproduced below:

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

The only subtlety has to do with the relationship between identifiers in the UL and variables 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 Γ 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:

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

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, Δ, and operate analagously (see supplement.)

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

Theorem 4.2 (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 covered in the next two subsections.

4.4 TLM Definitions

An unexpanded expression of seTLM definition form, $\texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for expressions by static}\ e_{\text{parse}}\ \texttt{in}\ \hat{e}$, defines an seTLM identified as $\hat{a}$ with *unexpanded type annotation* $\hat{\tau}$ and *parse function* e_{parse} for use within $\hat{e}$. The rule below defines typed expansion of this form:

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

The premises of this rule can be understood as follows, in order:

(1) The first premise expands the unexpanded type annotation.
(2) The second premise checks that e_{parse} is a closed expanded function[4] of type $\texttt{parr(Body;ParseResultSE)}$.
    The type abbreviated Body classifies encodings of literal bodies, b. The mapping from literal bodies to values of type Body is defined by the *body encoding judgement* $b \downarrow_{\text{Body}} e_{\text{body}}$. An inverse mapping is defined by the *body decoding judgement* $e_{\text{body}} \uparrow_{\text{Body}} b$. Rather than defining 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.)

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

The return type of the parse function, `ParseResultSE`, abbreviates a labeled sum type that distinguishes parse errors from successful parses:

$$L_{\mathsf{SE}} \stackrel{\text{def}}{=} \texttt{ParseError}, \texttt{SuccessE}$$

$$\texttt{ParseResultSE} \stackrel{\text{def}}{=} \texttt{sum}[L_{\mathsf{SE}}](\texttt{ParseError} \hookrightarrow \langle\rangle, \texttt{SuccessE} \hookrightarrow \texttt{PrExpr})$$

The type abbreviated `PrExpr` classifies encodings of *proto-expressions*, $\grave{e}$ (pronounced "grave e".) The syntax of proto-expressions, defined in Figure 8, 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 proto-expressions mediated by these judgements (see supplement.)

(3) 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 assuming an eagerly evaluated language.

(4) The final premise of Rule EE-DEF-SETSM extends the seTLM context, $\hat{\Psi}$, with the newly determined seTLM definition, and proceeds to assign a type, τ', and expansion, e, to $\hat{e}$. The conclusion of the rule then assigns this type and expansion to the seTLM definition as a whole.

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

The spTLM definition form, `syntax` $\hat{a}$ `at` $\hat{\tau}$ `for patterns by static` e_{parse} `in` $\hat{e}$, 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 share an identifier.

EE-DEF-SPTSM

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

4.5 TLM Application

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

EE-AP-SETSM

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

The premises of this rule can be understood as follows, in order:

(1) The first premise ensures that $\hat{a}$ has been defined and extracts the type annotation and parse function.
(2) The second premise determines the encoding of the literal body, e_{body}, which is a value of type `Body`.
(3) The third premise applies the parse function e_{parse} to the encoding of the literal body. If parsing succeeds, i.e. a value of the form `inj[SuccessE](`e_{proto}`)` results from evaluation, then e_{proto} will be a value of type `PrExpr` (assuming a well-formed seTLM context, by application of the Preservation assumption.) We call e_{proto} the *encoding of the proto-expansion*. If the parse function produces a value labeled `ParseError`, then typed expansion fails. Formally, no rule is necessary to handle this case.
(4) The fourth premise decodes the encoding of the proto-expansion to produce the proto-expansion itself.
(5) The fifth premise determines the segmentation, $\text{seg}(\grave{e})$, and ensures that it is valid with respect to b via the predicate ψ segments b, which checks that each segment in the finite set of segments ψ has non-negative length and is within bounds of b, and that the segments in ψ do not overlap.
(6) 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 discussed next.

The typed pattern expansion rule governing spTLM application is analogous:

PE-AP-SPTSM

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

4.6 Proto-Expansion Validation

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

Fig. 8. Syntax of miniVerse**s** proto-expansions. Proto-expansion terms are ABTs identified up to alpha-equivalence.

The *proto-expansion validation judgements* validate proto-expansions and simultaneously generate their final expansions:

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

The purpose of the *splicing scenes* $\mathbb{T}$, $\mathbb{E}$ and $\mathbb{P}$ is to "remember", during proto-expansion validation, the contexts and the 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.

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 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:

PTV-SPLICED

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

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*, $\langle \mathcal{D}; \Delta_{\mathrm{app}} \rangle$, but *not* the expansion-local type formation context, Δ. The final premise requires that the application site type formation context is disjoint from the expansion-local type formation context. Because proto-expansions are 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:

PEV-SPLICED

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

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 Δ and Γ) 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:

PPV-SPLICED

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

4.7 Metatheory

4.7.1 Typed Expression Expansion. Let us now return to Theorem 4.2, the typed expression expansion theorem which 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 4.3 (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 (spliced terms are distinguished only in the proto-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.

4.7.2 Typed Pattern Expansion. A similar argument is needed to prove Typed Pattern Expansion:

THEOREM 4.4 (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 $\dot{p} \rightsquigarrow p : \tau \dashv\Vdash^{\langle \mathcal{D}; \Delta \rangle; \langle \mathcal{A}; \Phi \rangle; b} \langle \mathcal{G}; \Gamma \rangle$ *then* $\Delta \vdash p : \tau \dashv\Vdash \Gamma$.

4.7.3 seTLM Reasoning Principles. The following theorem summarizes the abstract reasoning principles that programmers can rely on when applying an seTLM. Informal descriptions of important clauses are given inline.

THEOREM 4.5 (SETLM ABSTRACT 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) $b \downarrow_{\text{Body}} e_{body}$

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

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

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

(7) ***(Typing 2)*** $\{\langle \mathcal{D}; \Delta \rangle \vdash \text{parseUTyp}(\text{subseq}(b; m'_i; n'_i)) \rightsquigarrow \tau'_i\ \text{type}\}_{0 \le i < n_{ty}}$ *and* $\{\Delta \vdash \tau'_i\ \text{type}\}_{0 \le i < n_{ty}}$
Each spliced type has a well-formed expansion at the application site.

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

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

(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* $\{t_i\}_{0 \le i < n_{ty}}$ *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)*** $\text{fv}(e') \subset \{t_i\}_{0 \le i < n_{ty}} \cup \{x_i\}_{0 \le i < n_{exp}}$
The aforementioned decomposed term makes no mention of bindings in the application site context, i.e. the only free variables are those corresponding to the spliced terms.

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

4.7.4 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 4.6 (spTLM Abstract Reasoning Principles). *If* $\hat{\Delta} \vdash_{\hat{\Phi}} \hat{a}\ `b` \rightsquigarrow p : \tau \dashv\Vdash \hat{\Gamma}$ *where* $\hat{\Delta} = \langle \mathcal{D}; \Delta \rangle$ *and* $\hat{\Gamma} = \langle \mathcal{G}; \Gamma \rangle$ *then all of the following hold:*

(1) ***(Typing 1)*** $\hat{\Phi} = \hat{\Phi}', \hat{a} \rightsquigarrow a \hookrightarrow \texttt{sptlm}(\tau;\ e_{parse})$ *and* $\Delta \vdash p : \tau \dashv \Gamma$
(2) $b \downarrow_{\text{Body}} e_{body}$
(3) $e_{parse}(e_{body}) \Downarrow \texttt{inj[SuccessP]}(e_{proto})$
(4) $e_{proto} \uparrow_{\text{PrPat}} \grave{p}$
(5) ***(Segmentation)*** $\text{seg}(\grave{p})$ segments b
(6) $\text{seg}(\grave{p}) = \{\texttt{splicedt}[n_i'; m_i']\}_{0 \le i < n_{ty}} \cup \{\texttt{splicedp}[m_i; n_i; \grave{\tau}_i]\}_{0 \le i < n_{pat}}$
(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)*** $\{\emptyset \vdash^{\hat{\Delta}; b} \grave{\tau}_i \rightsquigarrow \tau_i\ \text{type}\}_{0 \le i < n_{pat}}$ *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) ***(No Hidden Bindings)*** $\mathcal{G} = \biguplus_{0 \le i < n_{pat}} \mathcal{G}_i$ *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.

The proof, in the supplement, relies on an auxiliary lemma about decomposing proto-patterns.

5 Parametric TLMs (pTLMs)

Simple TLMs operate at a single specified type and the proto-expansions they generate must be closed. This simplifies matters formally, but it is not convenient in practice. 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 ML signature specifying an abstract data type (Harper 1997; Liskov and Zilles 1974) of string-keyed polymorphic dictionaries:

```
signature DICT = sig
  type t('a)
  val empty : t('a)
  val extend : t('a) -> string * 'a -> t('a)
  (* ... *)
end
```

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

```
syntax $dict (D : DICT) (type 'a) at D.t('a) by static
  fn(b : body) -> parse_result(proto_expr) => (* ... *)
end
```

For example, assuming some implementation `Dict : DICT` and some values `v1 : int` and `v2 : int` (and assuming that VerseML offers a generalized literal form delimited by `{|` and `|}`), we can apply `$dict` as follows:

```
$dict Dict 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, so there can never be an ambiguity about where a key expression ends.)

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

```
let syntax $Dict = $dict Dict in
$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 `Dict` 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. In practice, the language might implicitly construct and open such a module within the expansion when the type annotation on the TLM is such a datatype.

It is important to note that module parameters are not accessible by parse functions directly, because the applied module parameters will not have been dynamically instantiated when typed expansion occurs. Parameters are accessible only from within the expansion that the parse function generates. 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 miniVerse$_\mathbf{P}$, a calculus that extends miniVerse$_\mathbf{S}$ with support for parametric TLMs. This calculus is organized, like miniVerse$_\mathbf{S}$, as an unexpanded language (UL) defined by typed expansion to an expanded language (XL). There is not enough space to describe miniVerse$_\mathbf{P}$ with the same level of detail as in Sec. 4, so we highlight only the most important concepts below.

The XL consists of 1) module expressions, M, classified by signatures, σ; 2) constructions, c, classified by kinds, κ; and 3) expressions classified by types, which are constructions of kind `Type` (we use metavariables τ instead of c for types.) 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 miniVerse$_\mathbf{S}$ only to allow projection out of modules.

The main conceptual difference between miniVerse$_\mathbf{S}$ and miniVerse$_\mathbf{P}$ is that miniVerse$_\mathbf{P}$ introduces the notion of unexpanded and expanded TLM expressions and types, as shown in Figure 9.

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

Fig. 9. Syntax of unexpanded and expanded TLM types and expressions in miniVerse$_\mathbf{P}$

This is necessary to express abstraction over parameters and application of parameters. For simplicity, we formalize only module parameters. Type parameters can be realized as module parameters having exactly one abstract type member.

The rule governing expression TLM application, reproduced below, touches all of the main ideas in miniVerse$_{\mathbf{P}}$.

EE-AP-PETSM

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

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

The fourth premise performs *peTLM expression normalization*. Normalization, $\Omega_{\text{app}} \vdash^{\text{Exp}}_{\Psi} \epsilon \Downarrow \epsilon_{\text{normal}}$, is defined in terms of a simple structural dynamics with two stepping rules:

EPS-DYN-APMOD-SUBST-E

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

EPS-DYN-APMOD-STEPS-E

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

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

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

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

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

There must be one binder in $\dot{e}$ for each TLM parameter specified by $\mathtt{tlmdef}(\epsilon_{\text{normal}})$. (VerseML can insert these binders automatically as a convenience, but we consider only the underlying mechanism in this core calculus.)

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

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

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

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

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

Proto-expansion validation operates conceptually as in miniVerse$_{\mathbf{S}}$. 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, ω, which is tracked by the splicing scene, must be applied to the type annotation before proceeding recursively to expand the referenced unexpanded term. However, the spliced term itself must treat parameters parametrically, so the substitution is not applied in the conclusion of the rule:

P-PEV-SPLICED

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

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

The calculus enjoys metatheoretic properties analagous to those described in Sec. 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.

Theorem 5.1 (peTLM Reasoning Principles). *If* $\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}$ ‘b‘ $\rightsquigarrow e : \tau$ *then:*

(1) $\hat{\Omega} = \langle \mathcal{M};\mathcal{D};\mathcal{G};\Omega_{app}\rangle$
(2) $\hat{\Psi} = \langle \mathcal{A};\Psi\rangle$
(3) ***(Typing 1)*** $\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{e} \rightsquigarrow \epsilon$ @ $\mathsf{type}(\tau)$ *and* $\Omega_{app} \vdash e : \tau$
(4) $\Omega_{app} \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \Downarrow \epsilon_{normal}$
(5) $\mathsf{tlmdef}(\epsilon_{normal}) = a$
(6) $\Psi = \Psi', a \hookrightarrow \mathsf{petlm}(\rho; e_{parse})$
(7) $b \downarrow_{\mathsf{Body}} e_{body}$
(8) $e_{parse}(e_{body}) \Downarrow \mathsf{inj}[\mathsf{SuccessE}](e_{pproto})$
(9) $e_{pproto} \uparrow_{\mathsf{PPrExpr}} \dot{e}$
(10) $\Omega_{app} \vdash^{\mathsf{Exp}}_{\Psi} \dot{e} \twoheadrightarrow_{\epsilon_{normal}} \grave{e}\ ?\ \mathsf{type}(\tau_{proto}) \dashv \omega : \Omega_{params}$
(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) $\mathsf{seg}(\grave{e}) = \{\mathsf{splicedk}[m_i;n_i]\}_{0\le i<n_{kind}} \cup \{\mathsf{splicedc}[m'_i;n'_i;\grave{\kappa}'_i]\}_{0\le i<n_{con}} \cup \{\mathsf{splicede}[m''_i;n''_i;\grave{\tau}_i]\}_{0\le i<n_{exp}}$
(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)*** $\text{fv}(e'') \subset \{k_i\}_{0 \le i < n_{kind}} \cup \{u_i\}_{0 \le i < n_{con}} \cup \{x_i\}_{0 \le i < n_{exp}} \cup dom(\Omega_{params})$

We leave out the details of parametric pattern TLMs (ppTLMs), but they should again follow analagously.

6 Static Evaluation

Thus far, we have assumed that the parse functions in TLM definitions 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 need to introduce the concept of a *static environment*.

6.1 The Static Environment

Figure 10 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 the parse functions of the subsequent TLM definitions.

```
static module ParserCombos = struct
  type parser('c, 't) = list('c) -> list('t * list('c))
  val alt : parser('c, 't) -> parser('c, 't) -> parser('c, 't)
  (* ... *)
end
syntax $a at t by static fn(b) => (* OK *) ParserCombos.alt (* ... *) end
static val y = (* OK *) ParserCombos.alt (* ... *)
val z = (* neither ParserCombos nor y are available here *)
```

Fig. 10. Binding a static module for use within parse functions.

The values that arise during the static evaluation (in particular, of parse functions) do not themselves 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 at the end of 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 in a disciplined manner, 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 fn(b) => (* ... *) end
```

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:

```
signature PARSEGEN = sig
  type grammar('a)
  (* ... operations on grammars ... *)
  val generate : grammar('a) -> (body -> parse_result('a))
end
```

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 `$grammar`:

```
static syntax $grammar (P : PARSEGEN) 'a at P.grammar('a) by (* ... *) end
```

To support splicing, we need non-terminals that recognize unexpanded terms and produce the corresponding splice reference, rather than the AST itself. This would require that the parser generator keep track of location information (as most production-grade parser generators already do for error reporting.) For spliced expressions, this non-terminal would need to be a family of non-terminals indexed by a value of type `proto_typ`.

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 splice summary 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 generated proto-expansion does in fact mention each spliced expression from the corresponding production.

6.3 Static Evaluation, Formally

It is not difficult to extend miniVerse$_{\mathbf{P}}$ to account for static evaluation. Static environments, Σ, take the form $\omega : \hat{\Omega}; \hat{\Psi}; \hat{\Phi}$, where ω is a substitution. Each binding form is annotated with a *phase*, ϕ, either `static` or `standard`. The rules for binding forms annotated with `standard` are essentially unchanged, differing only in that Σ 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 Σ 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 ω is applied to the parse function after it is expanded. The full details are defined as a minor patch of miniVerse$_{\mathbf{P}}$ called miniVerse$_{\mathbf{PH}}$ 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 a new instance of a module imported into 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). We will leave these details as design decisions for the reader to consider.

TLMs definitions can be exported by packages, but they cannot be exported from within 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. It would also bring in confusion about whether the generated expansions can use knowledge about type identity available within the module enclosing the TLM definition. 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.

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

Fig. 11. An example of unreasonable program text.

7 REASONING CRITERIA AND EXISTING APPROACHES

7.1 Library-Specific Syntax Dialects

One approach available to library providers 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. Ur/Web's syntax (Fig. 1) is a library-specific dialect of Ur's syntax.

There are hundreds of syntax definition systems of various design, some integrated deeply into language front-ends and build systems, others operating as standalone preprocessors. Examples include grammar-based systems like Camlp4 (Leroy et al. 2014), Copper (Wyk and Schwerdfeger 2007) or SugarJ/Sugar* (Erdweg et al. 2011; Erdweg and Rieger 2013), parser combinator systems (Hutton 1992) and less general operator-based systems, like the system of fixity directives in Standard ML (Milner et al. 1997) and the more elaborate mixfix systems (Griffin 1988; Taha and Johann 2003) built in to proof assistants like Agda (Danielsson and Norell 2008) and Coq (Coq Development Team 2004).

er citation?

er citation?

Criteria 1: Modular Reasoning About Syntactic Determinism. The first problem with this approach is that clients cannot always deterministically combine different library-specific syntax dialects when they want to use the new forms that they define together. 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".

In some cases, the dialects in question cannot be combined simply because they have been defined under incompatible syntax definition systems. In other cases, the problem is one of *conflict*: two dialects might define the same form, but determine different desugarings. For example, consider two syntax dialects defined under a system like Camlp4: $\mathcal{D}_1$ defines literal forms for sets, and $\mathcal{D}_2$ defines literal forms for finite sets, both delimited by {< and >}. Each dialect is deterministic (i.e. there are no possible syntactic ambiguities), denoted $\det(\mathcal{D}_1)$ and $\det(\mathcal{D}_2)$. However, when the grammars are combined by Camlp4, it is not the case that $\det(\mathcal{D}_1 \cup \mathcal{D}_2)$ because {<>} can desugar to the empty set or the empty dictionary. Reasoning about determinism is therefore not modular.

Schwerdfeger and Wyk (2009) have developed a modular grammar analysis, implemented in Copper, that guarantees that determinism is conserved when syntax dialects of a certain restricted class are combined, the caveat being 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 (as one might do when, for example, deeply nested modules are used for namespacing in ML.)

Criteria 2: Abstract Reasoning About Literal Forms. 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. For example, consider the perspective of a programmer attempting to comprehend (i.e. reason about) the program fragment in Figure 11, which is written in a syntax dialect constructed by combining various library-specific dialects of ML's (or Scala's) syntax. If the programmer happens to be familiar with the (intentionally terse) syntax of the stack-based database query processing language K (Whitney and Shasha 2001), then Line 3 might pose few difficulties. If the programmer does not recognize this syntax, however, there are no simple, definitive protocols for answering questions like:

(1) **(Responsibility)** Which constituent dialect defined the literal form that appears on Line 3?

(2) **(Segmentation)** Are the characters `x` and `R` on Line 3 parsed as spliced expressions `x` and `R` (i.e. expressions of variable form), or parsed in some other way peculiar to this literal form?
(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 introduced in the desugaring of Line 3?
(4) **(Context Dependence)** If `w`, on Line 1, is renamed, could that possibly break Line 3, or change its meaning? In other words, might the desugaring of Line 3 assume that some variable identified as `w` is in scope (even though `w` is not mentioned in the text of Line 3)?
(5) **(Typing)** What type does `y` have?

In short, syntax dialects do not come equipped with principles of *syntactic abstraction*: if the desugaring of the program text is held abstract, programmers can no longer reason about types, binding and segmentation (i.e. answer questions like those above) in the usual disciplined manner. This is burdensome at all scales, but particularly when programming in the large, where it is common to encounter unfamiliar library constructs. Forcing the programmer to examine the desugaring of the program text, or the logic that produced that desugaring, in order to answer these sorts of basic questions about segmentation, binding and typing can defeat the ultimate purpose of syntactic sugar – decreasing cognitive cost (Green 1989).

7.2 Another Existing Approach: Term Rewriting

An alternative approach (and the approach that our mechanism is rooted in) is to leave the context-free syntax of the language unmodified and instead contextually repurpose existing forms using a *term rewriting system*.

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 attempt to mark a string literal containing Ur/Web-style HTML syntax with a ppx annotation, `xml`, as follows:

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

Again, 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 non-determinism can arise, and it is difficult to reason about **Responsibility**. It is also difficult to reason abstractly about **Segmentation**, **Capture**, **Context Dependence** and **Typing** because the code that the preprocessor generates is arbitrary. For these reasons, users of this mechanism warn that they should be used sparingly. citation

Term-rewriting macro systems address these problems – the applied macro explicitly identifies the rewriting that wil

Template Haskell / Haskell quasiquotations / early macro systems / reader macros
hygienic macro systems – problem!
typed macro systems, e.g. Scala. mention MacroML stuff, Herman and Wand 2008/2010 work
string splicing / interpolation
TSLs / TSMs

Non-local rewriting systems suffer from problems analagous to those that plague syntax definition systems. In particular, there can be conflicts because separately defined rewriting rules might attempt to rewrite the same term differently. It can also be difficult to determine which rewriting rule, if any, is responsible for any particular term. Certain conventions have emerged, e.g. OCaml's textual syntax now includes *preprocessor extension (ppx) points* used to identify terms that some presumably corresponding term rewriting preprocessor must rewrite (Leroy et al. 2014), but a correspondence is not enforced. In any case, it is also difficult to reason abstractly about types and binding given a program subject to a large number of rewriting rules.

Modern *term-rewriting macro systems*, however, have made some progress. In particular:

(1) Macro systems require that the client explicitly apply the intended rewriting (implemented by a macro) to the term that is to be rewritten, thereby addressing the problems of conflict and determining responsibility.

However, it is often unclear whether a given macro is repurposing the form of a given argument or sub-term thereof, as opposed to treating it parametrically by inserting it unmodified into the generated expansion. This is closely related to the problem of determining a segmentation discussed above.

(2) *Hygienic* macro systems prevent variable capture and enforce context-independence (Adams 2015; Clinger and Rees 1991; Dybvig et al. 1992; Kohlbecker et al. 1986).

(3) Although much of the research on macro systems has been for languages in the Lisp tradition (McCarthy 1978), some progress has also been made on reasoning about types with the design of *typed macro systems* where annotations constrain the macro arguments and the generated expansions. Examples include Scala's macro system (Burmako 2013), formal calculi studied by Herman and Wand (2008) and *staging macro systems* like MetaML (Sheard 1999) and MacroML (Ganz et al. 2001) (which, it is worth noting, do not give the macro access to the syntax tree of arguments.)

Perhaps the biggest problem, then, is that term-rewriting systems offer library providers only limited syntactic control. In other words many syntactic forms of interest will not particularly resemble those available in the host language. For example, consider the XHTML and K examples above. In both cases, the syntactic conventions are quite distinct from those of ML-like languages (and, for that matter, languages that use S-expression.)

It is tempting in these situations to consider repurposing string literal forms. For example, we might wish to apply a macro `html!` (following Rust's convention of using a post-fix `!` to distinguish macro names from variables) to rewrite string literals containing Ur/Web-style XHTML syntax as follows:

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

The problem here is that there is no way to extract the spliced expressions from the supplied string literal forms while satisfying the hygiene conditions, because variables that come from these spliced terms (here, `first_name` and `heading`) are indistinguishable from variables that inappropriately appear free relative to the expansion. In addition, the problem of segmentation becomes even more pernicious: to a human or tool unaware of Ur/Web's syntax, it is not immediately apparent which particular subsequences of the string literals supplied to `html!` are segmented out as spliced expressions. Reader macros have essentially the same problem (Flatt et al. 2012).

Some languages build in *string splicing* (a.k.a. *string interpolation*) forms, e.g. Scala (Odersky et al. 2008), or similar but more general *fragmentary quotation forms* (Slind 1991), e.g. the SML/NJ dialect of ML. These designate particular delimiters to allow escape out into the expression language. The problem with using these (dynamically, or in conjunction with term rewriting macros) as vehicles to introduce syntactic sugar at various types is 1) there is no "one-size-fits-all" escape delimiter, and 2) reasoning about typing is problematic because every escaped term is checked against the same type. In the example above, we spliced in two different types of expressions using two different delimiters. In general, e.g. when defining syntax for a programming language with many sorts of terms, the most appropriate choice of delimiters might depend on where each spliced term appears.

8 Additional Related Work & Concluding Discussion

The most closely related work is that of Omar et al. (2014) on *type-specific languages* (TSLs), which introduced generalized literal forms and gave a bidirectionally typed protocol for type-directed parse function invocation. This work differs in several ways, and the differences we note below nicely summarize our specific contributions.

Most obviously, TLMs are explicitly applied. Different TLMs can therefore be defined at the same type without conflict. Moreover, TLMs are not associated directly with generative type definitions, like TSLs, but rather can operate at any type. In a subsequent short paper, Omar et al. (2015) proposed explicit application of simple expression TLMs in a bidirectional typed setting (Pierce and Turner 2000), but did not provide any formal details. We do not assume a bidirectionally typed language. Context independence implies that ML-style type inference can be performed using only the spliced terms and their types (because the remainder of an expansion

cannot mention the very variables whose types are being inferred, nor constrain them further beyond what the annotations require.) We leave additional inference-related topics, e.g. TLM parameter inference, as future work.

Another distinction is that the metatheory presented by Omar et al. (2014) establishes only that generated expansions are well-typed. It does not establish the remaining abstract reasoning principles that have been a major focus of this paper. In particular, there is no formal notion of hygiene (though it is discussed informally), and the mechanism does not guarantee that a valid segmentation will exist, nor associate types with segments.

Finally, the prior work on TSLs has not considered pattern matching, parameterized types, modules, parametric expansions or static evaluation. All of these are important for integration into an ML-like functional language.

Another particularly compelling stream of related work is that by Lorenzen and Erdweg (2013, 2016), which describes 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.

Our work differs in several ways. First, 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. Relatedly, TLMs do not support type-dependent expansion, as in (Lorenzen and Erdweg 2016). Another important distinction is that our system relies on (proto-)expansion validation, rather than verification. In other words, we do not prove properties about the parse function itself, but rather "test" each expansion. The trade-off is that our system does not require that parsing logic be written in a restricted manner, nor require a fully mechanized language definition and 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".

One important avenue has to do with automated refactoring. The unexpanded language does not come equipped with standard notions of renaming and substitution. Given a splice summary, it should be possible to "backpatch" refactorings into literal bodies, but the details are left as future work.

Haskell's **do**-notation for types with monadic structure cannot be expressed directly using TLMs because it introduces bindings into sub-terms. This might be an acceptable trade-off (indeed, that is our present position.) Alternatively, it might be worth exploring a design where spliced identifiers (but not arbitrary identifiers) can be designated in the splice summary as bound within spliced terms.

At several points in the paper, we discussed editor integration. However, some important questions having to do with incremental parsing, error recovery and error reporting remain to be considered. Proto-expansions are context-independent and parametric, so they are likely to amenable to caching.

Another interesting question is what this mechanism would look like in the setting of a projectional structure editor, i.e. one where the syntax is not textual but rather tree-shaped and projected in an interative manner to the programmer. This paper taken together with work by Omar et al. (2017, 2012) can serve to guide such an effort.

To conclude, TLMs give substantial syntactic control to library providers while leaving programmers with strong abstract reasoning principles. We believe TLMs therefore occupy a "sweet spot" in the design space.

REFERENCES

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

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

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

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

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

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

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, Ryan Culpepper, David Darais, and Robert Bruce Findler. 2012. Macros that Work Together - Compile-time bindings, partial expansion, and definition contexts. *J. Funct. Program.* 22, 2 (2012), 181–216. http://dx.doi.org/10.1017/S0956796812000093

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

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

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

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

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

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

David Herman and Mitchell Wand. 2008. A Theory of Hygienic Macros. In *ESOP*. http://dx.doi.org/10.1007/978-3-540-78739-6_4

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

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

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

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

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)*.

Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers. 2012. Active Code Completion. In *International Conference on Software Engineering (ICSE)*. http://dl.acm.org/citation.cfm?id=2337223.2337324

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.

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

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

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

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

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.

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

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

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

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

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