Reasonably Programmable Syntax

Cyrus Omar

April 28, 2017

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jonathan Aldrich, Chair
Robert Harper
Karl Crary
Eric Van Wyk, University of Minnesota

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

TODO: Support

Keywords: syntax, parsing, type systems, module systems, macro systems, hygiene, pattern matching, bidirectional typechecking, implicit dispatch, metaprogramming

TODO: dedication

Abstract

Programming languages commonly provide "syntactic sugar" that decreases the syntactic cost of working with certain standard library constructs. For example, Standard ML builds in syntactic sugar for constructing and pattern matching on lists. Third-party library providers are, justifiably, envious of this special arrangement. After all, it is not difficult to find other examples of situationally useful library-specific syntactic sugar [96]. For example, 1) clients of a "collections" library might like syntactic sugar for finite sets and dictionaries; 2) clients of a "web programming" library might like syntactic sugar for HTML and JSON values; 3) a compiler writer might like syntactic sugar for the terms of the object language or various intermediate languages of interest; and 4) clients of a "chemistry" library might like syntactic sugar for chemical structures based on the SMILES standard [16].

Defining a "library-specific" syntax dialect in each of these situations is problematic, because library clients cannot combine dialects like these in a manner that conserves syntactic determinism (i.e. syntactic conflicts can and do arise.) Moreover, it can become difficult for library clients to reason abstractly about types and binding when examining the text of a program that uses unfamiliar forms. Typed, hygienic term-rewriting macro systems are more reasonable, but offer only limited syntactic control.

This work formally introduces *typed syntax macros (TSMs)*, which reduce the need for library-specific syntax dialects by giving library providers the ability to programmatically control the parsing and expansion, at "compile-time", of expressions and patterns of *generalized literal form*. Library clients can use any combination of TSMs in a program without needing to consider the possibility of syntactic conflict, because the context-free syntax of the language is never extended (rather, it is contextually repurposed.) Moreover, the language validates each expansion that a TSM generates in order to maintain useful abstract reasoning principles. In particular, expansion validation allows the TSM client to maintain:

- a *type discipline*, meaning that the client can reason about types while holding literal expansions abstract; and
- a *hygienic binding discipline*, meaning that the client can be sure that:
  1. literal expansions cannot shadow bindings that appear at the TSM application site; and
  2. literal expansions do not refer to definition-site or application-site bindings directly. Instead, all interactions with bindings external to the expansion go through either explicit *spliced terms* or *parameters*.

In short, we describe a programming language (in the ML tradition) with a *reasonably* programmable syntax.

Acknowledgments

TODO: acknowledgments

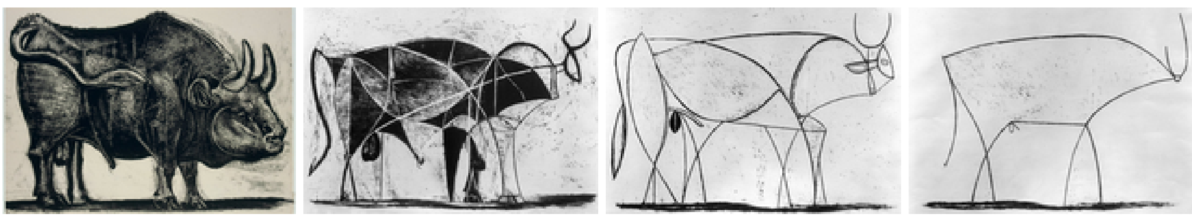
Contents

List of Figures

DRAFT (April 28, 2017)

Chapter 1

Introduction



Pablo Picasso (1881-1973)

1.1 Motivation

Experienced mathematicians and programmers define formal structures *compositionally*, drawing from libraries of "general-purpose" abstractions. The problem that motivates this work is that the resulting terms are sometimes syntactically unwieldy, and, therefore, cognitively costly.

Consider, for example, natural numbers. It is straightforward to define the natural numbers, n, as an inductive structure:

$$n ::= \mathbf{z} \mid \mathbf{s}(n)$$

By defining natural numbers inductively, we immediately inherit a *structural induction principle* – we can establish that some property P holds over the natural numbers if we establish $P(\mathbf{z})$ and $P(\mathbf{s}(n))$ assuming $P(n)$. The problem, of course, is that drawing particular natural numbers by repeatedly applying $\mathbf{s}$ very quickly becomes syntactically unwieldy (in fact, the syntactic cost of the drawing grows linearly with n.)[1]

Similarly, it is easy to define lists of natural numbers as an inductive structure:

$$\vec{n} ::= \mathbf{nil} \mid \mathbf{cons}(n, \vec{n})$$

The problem once again is that drawings of particular lists quickly become unwieldy, and fail to resemble "naturally occurring" drawings of lists of numbers.

[1]We use the word "drawing" throughout this document to emphasize that syntactic cost is a property of the visual representation of a structure, rather than a semantic property.

Sort			Operational Form	Stylized Form	Textual Form	Description
CalcExp	e	$::=$	x	x	x	variable
			$\texttt{let}(e; x.e)$	$\texttt{let}\ x = e\ \texttt{in}\ e$	$\texttt{let}\ x = e\ \texttt{in}\ e$	binding
			$\texttt{num}[n]$	$\texttt{n}$	$\texttt{n}$	numbers
			$\texttt{plus}(e; e)$	$e + e$	e `+` e	addition
			$\texttt{mult}(e; e)$	$e \times e$	e `*` e	multiplication
			$\texttt{div}(e; e)$	$\frac{e}{e}$	e `/` e	division
			$\texttt{pow}(e; e)$	e^e	e`^`e	exponentiation

Figure 1.1: Syntax of **Calc**. Metavariable n ranges over natural numbers and $\texttt{n}$ abbreviates the numeral forms (one for each natural number n, drawn in `typewriter` font.) A formal definition of the stylized and textual syntax of **Calc** would require 1) defining these numeral forms explicitly; 2) defining a parenthetical form; 3) defining the precedence and associativity of each infix operator; and 4) defining whitespace conventions.

Consider a third more sophisticated example (which will be of particular relevance later in this work): when defining a programming language or logic, one often needs various sorts of tree structures equipped with metaoperations[2] related to variable binding, e.g. substitution. Repeatedly defining these structures "from scratch" is quite tedious, so language designers have instead developed a more general structure: the *abstract binding tree (ABT)* [14, 52, 61]. Briefly, an ABT is an ordered tree structure, classified into one of several *sorts*, where each node is either a *variable*, x, or an *operation* of the following form:

$$\texttt{op}(\vec{x}_1.a_1; \ldots; \vec{x}_n.a_n)$$

where $\texttt{op}$ identifies an *operator* and each of the $n \geq 0$ *arguments* $\vec{x}_i.a_i$ binds the (possibly empty) sequence of variables $\vec{x}_i$ within the subtree a_i. The left side of the syntax chart in Figure 1.1 summarizes the relevant operational forms for a sort called CalcExp. ABTs of this sort are the expressions of a small arithmetic programming language, **Calc**. By using ABTs as infrastructure in the definition of **Calc**, we need not manually define the "boilerplate" metaoperations, like substitution, and reasoning principles, like structural induction, that are necessary to define **Calc**'s semantics and to prove it correct. Harper gives a detailed account of ABTs, and many other examples of their use, in his book [61].

The problem with this approach is, again, that drawing a non-trivial **Calc** expression in operational form is syntactically costly. For example, we will consider the following drawing in our discussion below:

$$\texttt{div}(\texttt{num}[\mathbf{s}(\mathbf{z})];\texttt{pow}(\texttt{num}[\mathbf{s}(\mathbf{s}(\mathbf{z}))];\texttt{div}(\texttt{num}[\mathbf{s}(\mathbf{z})];\texttt{num}[\mathbf{s}(\mathbf{s}(\mathbf{z}))]))) \tag{1.1a}$$

1.1.1 Informal Mathematical Practice

Within a document intended only for human consumption, it is easy to informally outline less costly alternative syntactic forms.

[2]...so named to distinguish them from the "object level" operations of the language being defined.

For example, mathematicians generally use the Western Arabic numeral forms when drawing particular natural numbers, e.g. 2 is taken as a syntactic alternative to $\mathbf{s}(\mathbf{s}(\mathbf{z}))$.

Similarly, mathematicians might informally define alternative list forms, e.g. $[0, 1, 2]$ as a syntactic alternative to:

$$\mathbf{cons}(\mathbf{z}, \mathbf{cons}(\mathbf{s}(\mathbf{z}), \mathbf{cons}(\mathbf{s}(\mathbf{s}(\mathbf{z})), \mathbf{nil})))$$

The middle columns of the syntax chart in Figure 1.1 suggest two alternative forms for every ABT of sort CalcExp. We can draw the ABT from Drawing (1.1a) in an alternative *stylized form*:

$$\frac{1}{2^{\frac{1}{2}}} \tag{1.1b}$$

or in an alternative *textual form*:

$$\texttt{1 / 2^(1/2)} \tag{1.1c}$$

Mathematicians also sometimes supplement alternative primitive forms like these with various *derived forms*, which identify ABTs indirectly according to stated context-independent *desugaring rules*. For example, the following desugaring rule defines a derived stylized form for square root calculations:

$$\sqrt{e} \rightharpoonup e^{\frac{1}{2}} \tag{1.2}$$

The reader can desugar a drawing of an ABT by recursively applying desugaring rules wherever a syntactic match occurs. A desugared drawing consists only of the primitive forms from Figure 1.1. For example, the following drawing desugars to Drawing (1.1b), which in turn corresponds to Drawing (1.1a) as discussed above:

$$\frac{1}{\sqrt{2}} \tag{1.1d}$$

When defining the semantics of a language like **Calc**, it is customary to adopt an *identification convention* whereby drawings that identify the same underlying ABT structure, like Drawings (1.1), are considered interchangeable. For example, consider the semantic judgement e val, which establishes certain **Calc** expressions as *values* (as distinct from expressions that can be arithmetically simplified or that are erroneous.) The following inference rule establishes that every number expression is a value:[3]

$$\frac{}{\texttt{num}[n]\ \mathsf{val}} \tag{1.3}$$

Although this rule is drawn using the operational form for number expressions, we can apply it to derive that 2 val, because 2 and $\texttt{num}[2]$ identify the same ABT.

[3]Some familiarity with inductively defined judgements and inference rules like these is preliminary to this work. See Sec. 2.1 for citations and further discussion of necessary preliminaries.

1.1.2 Derived Forms in General-Purpose Languages

We would need to define only a few more derived arithmetic forms to satisfyingly capture the idioms that arise in the limited domains where a simple language of arithmetic operations like **Calc** might be useful. However, programming languages in common use today are substantially more semantically expressive. Indeed, many mathematical structures, including natural numbers, lists and ABTs, can be adequately expressed within contemporary "general-purpose" programming languages. Consequently, the problems of syntactic cost just discussed at the level of the ambient mathematics also arise "one level down", i.e. when writing programs. For example, we want syntactic sugar not only for mathematical natural numbers, lists and **Calc** expressions, but also for *encodings* of these structures within a general-purpose programming language.

We can continue to rely on the informal notational conventions described above only as long as programs are drawn solely for human consumption. These conventions break down when we need drawings of programs to themselves exist as formal structures suitable for consumption by other programs, i.e. *parsers*, which check whether drawings are well-formed relative to a *syntax definition* and produce structures suitable for consumption by yet other programs, e.g. compilers.

Constructing a formal syntax definition is not itself an unusually difficult task for an experienced programmer, and there are many *syntax definition systems* that help with this task (Sec. 2.4 will cover several examples.) The problem is that when designing the syntax of a general-purpose language, the language designer cannot hope to anticipate all library constructs for which derived forms might one day be useful. At best, the language designer can bundle certain libraries together into a "standard library", and privilege select constructs defined in this library with derived forms.

For example, the textual syntax of Standard ML (SML), a general-purpose language in the functional tradition, defines derived forms for constructing and pattern matching on lists [59, 89]. In SML, the derived expression form `[x, y, z]` desugars to an expression equivalent to:

```
Cons(x, Cons(y, Cons(z, Nil)))
```

assuming `Nil` and `Cons` stand for the list constructors exported by the SML Basis library (i.e. SML's "standard library".)[4] Other languages similarly privilege select standard library constructs with derived forms:

- OCaml [79] defines derived forms for strings (defined as arrays of characters.)
- Haskell [72] defines derived forms for encapsulated commands (and, more generally, values of any type equipped with monadic structure.)
- Scala [93] defines derived XML forms as well as string splicing forms, which capture the idioms of string concatenation.
- F# [118], Scala [108] and various other languages define derived forms for encodings of the language's own terms (these are referred to as *quasiquotation* forms.)

[4]The desugaring actually uses unforgeable identifiers bound permanently to the list constructors, to ensure that the desugaring is context independent. We will return to the concept of context independence throughout this work.

- Python [9] defines derived forms for mutable sets and dictionaries.
- Perl [8] defines derived regular expression forms.

These choices are, fundamentally, made according to *ad hoc* design criteria – there are no clear semantic criteria that fundamentally distinguish standard library constructs privileged with derived forms from those defined in third-party libraries. Indeed, as the OCaml community has moved away from a single standard library in favor of competing bundles of third-party libraries (e.g. Batteries Included [3] and Core [4]), this approach has become starkly impractical.

1.2 Existing Mechanisms of Syntactic Control

A more parsimonious approach would be to eliminate derived forms specific to standard library constructs from language definitions in favor of mechanisms that give more syntactic control to third-party library providers.

In this section, we will give a brief overview of existing such mechanisms and speak generally about the problems that they present to motivate our novel contributions in this area. We will return to give a detailed overview of these various existing mechanisms of syntactic control in Section 2.4.

1.2.1 Syntax Dialects

One approach that a library provider can take when seeking more syntactic control is to use a syntax definition system to construct a *syntax dialect*, i.e. a new syntax definition that extends the original syntax definition with new derived forms.

For example, Ur/Web extends Ur's textual syntax with derived forms for SQL queries, XHTML elements and other constructs defined in a web programming library [25, 26]. Figure 1.2 demonstrates how XHTML expressions that contain strings can be drawn in Ur/Web. The desugaring of this derived form (not shown) is substantially more verbose and, for programmers familiar with the standardized syntax for XHTML [127], substantially more obscure.

```
val p = <xml><p>Hello, {[join " " [first, last]]}!</p></xml>
```

Figure 1.2: Derived XHTML forms in Ur/Web

Syntax definition systems like Camlp4 [79], Copper [131] and SugarJ/Sugar* [41, 43], which we will discuss in Sec. 2.4.5, have simplified the task of defining "library-specific" (a.k.a. "domain-specific") syntax dialects like Ur/Web, and have thereby contributed to their ongoing proliferation.

Many have argued that a proliferation of syntax dialects is harmless or even desirable, because programmers can simply choose the right syntax dialect for each job at hand [128]. However, we argue that this "dialect-oriented approach" is difficult to reconcile with the best practices of "programming in the large" [35], i.e. developing large programs "consisting of many small programs (modules), possibly written by different people" whose interactions are mediated by a reasonable type and binding discipline. The problems that tend to arise are summarized below; a more systematic treatment will follow in Sec. 2.4.5.

Problem 1: Conservatively Combining Syntax Dialects

The first problem with the dialect-oriented approach is that clients cannot always combine different syntax dialects when they want to use derived forms that they define together. This is problematic because client programs cannot be expected to fall cleanly into a single preconceived "problem domain" – large programs use many libraries [77].

For example, consider a syntax dialect, $\mathcal{H}$, defining derived forms for working with encodings of HTML elements, and another syntax dialect, $\mathcal{R}$, defining derived forms for working with encodings of regular expressions. Some programs will undoubtedly need to manipulate HTML elements as well as regular expressions, so it would be useful to construct a "combined dialect" where all of these derived forms are defined.

For this notion of "dialect combination" to be well-defined at all, we must first have that $\mathcal{H}$ and $\mathcal{R}$ are defined under the same syntax definition system. In practice, there are many useful syntax definition systems, each differing subtly from the others.

If $\mathcal{H}$ and $\mathcal{R}$ are coincidentally defined under the same syntax definition system, we must also have that this system operationalizes the notion of dialect combination, i.e. it must define some operation $\mathcal{H} \cup \mathcal{R}$ that creates a dialect that extends both $\mathcal{H}$ and $\mathcal{R}$, meaning that any form defined by either $\mathcal{H}$ or $\mathcal{R}$ must be defined by $\mathcal{H} \cup \mathcal{R}$. Under systems that do not define such an operation (e.g. Racket's dialect preprocessor [48]), clients can only manually "copy-and-paste" or factor out portions of the constituent dialect definitions to construct the "combined" dialect. This is not systematic and, in practice, it can be quite tedious and error-prone.

Even if we restrict our interest to dialects defined under a common syntax definition system that does operationalize the notion of dialect combination (or similarly one that allows clients to systematically combine *dialect fragments*), we still have a problem: there is generally no guarantee that the combined dialect will conserve important properties that can be established about the constituent dialects in isolation (i.e. *modularly*.) In other words, establishing $P(\mathcal{H})$ and $P(\mathcal{R})$ is not sufficient to establish $P(\mathcal{H} \cup \mathcal{R})$ for many useful properties P. Clients must re-establish such properties for each combined dialect that they construct.

One important property of interest is *syntactic determinism* – that every derived form has at most one desugaring. It is not difficult to come up with examples where combining two deterministic syntax dialects produces a non-deterministic dialect. For example, consider two syntax dialects defined under a system like Camlp4: $\mathcal{D}_1$ defines derived forms for sets, and $\mathcal{D}_2$ defines derived forms for finite maps, both delimited by `{<` and `>}`.[5] Though each dialect defines a deterministic grammar, i.e. $\det(\mathcal{D}_1)$ and $\det(\mathcal{D}_2)$,

[5] In OCaml, simple curly braces are already reserved by the language for record types and values.

when the grammars are naïvely combined by Camlp4, we do not have that $\mathrm{det}(\mathcal{D}_1 \cup \mathcal{D}_2)$ (i.e. syntactic ambiguities arise under the combined dialect.) In particular, `{<>}` can be recognized as either the empty set or the empty finite map.

Schwerdfeger and Van Wyk have developed a modular grammar-based syntax definition system, implemented in Copper [131], that guarantees that determinism is conserved when syntax dialects (of a certain restricted class) are combined [106] as long as each constituent dialect prefixes all newly introduced forms with starting tokens drawn from disjoint sets. We will describe the difficulties that this requirement causes in Section 2.4.5.

Problem 2: Abstract Reasoning About Derived Forms

Even putting aside the difficulties of conservatively combining syntax dialects, there are questions about how *reasonable* sprinkling library-specific derived forms throughout a large software system might be. For example, consider the perspective of a programmer attempting to comprehend (i.e. reason about) the program fragment in Figure 1.3, which is drawn under a syntax dialect constructed by combining a number of dialects of Standard ML's textual syntax.

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

Figure 1.3: An example of unreasonable program text

If the programmer happens to be familiar with the (intentionally terse) syntax of the stack-based database query processing language K [129], 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 derived 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 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 the program, 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 with useful principles of *syntactic abstraction*: if the desugaring of the program is held abstract, programmers can no longer reason about types and binding (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 a program fragment drawn by another programmer, or drawn long ago. Forcing the programmer to examine the desugaring of the drawing in order to reason about types and binding defeats the ultimate purpose of using syntactic sugar – lowering cognitive cost (we expand on the notion of cognitive cost in Sec. 2.2.)

In contrast, when a programmer encounters, for example, a function call like the call to `compute_x` on Line 3, the analagous questions can be answered by following clear protocols that become "cognitive reflexes" after sufficient experience with the language, even if the programmer has no experience with the library defining `compute_x`:

1. The language's syntax definition determines that `compute_x w` is an expression of function application form.
2. Similarly, `compute_x` and `w` are definitively expressions of variable form.
3. The variable `w` can only refer to the binding of `w` on Line 1.
4. The variable `w` can be renamed without knowing anything about the value that `compute_x` stands for.
5. The type of `x` can be determined to be `B` by determining that the type of `compute_x` is `A -> B` for some `A` and `B`, and checking that `w` has type `A`. Nothing else needs to be known about the value that `compute_x` stands for. In Reynolds' words [104]:

   Type structure is a syntactic discipline for enforcing levels of abstraction.

1.2.2 Term Rewriting Systems

An alternative approach that a library provider can consider when seeking to control syntactic cost is to leave the context-free syntax of the language fixed and instead contextually repurpose existing syntactic forms using a *term rewriting system*. We will review various term rewriting systems in detail in Sec. 2.4.9 and Sec. 2.4.10.

Naïve term rewriting systems suffer from problems analagous to those that plague syntax definition systems. In particular, it is difficult to conserve determinism, i.e. separately defined rewriting rules might attempt to rewrite the same term differently. Moreover, it can be difficult to determine which rewriting rule, if any, is responsible for a particular term, and to reason about types and binding given a drawing of a program subject to a large number of rewriting rules without examining the rewritten program.

Modern *term-rewriting macro systems*, however, have made some progress toward addressing these problems. 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. Macro systems that enforce *hygiene*, which we will return to in Sec. 2.4.10, address many of the problems related to reasoning about binding.
3. The problem of reasoning about types has been relatively understudied, because most research on macro systems has been for languages in the Lisp tradition that lack rich static type structure [87]. That said, some progress has also been made on this front with the design of *typed macro systems*, like Scala's macro system [23], where annotations constrain the macro arguments and the generated expansions.

The main problem with term-rewriting macros, then, is that they afford library providers only limited syntactic control – they must find creative ways to repurpose existing forms. 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:

```
html! "<p>Hello, {[join " " [first, last]]}!</p>"
```

The problem here is that there is no way to extract the spliced expressions from the supplied string literal forms while satisfying the context independence condition, because variables that come from these spliced terms (e.g. `join`) 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 [49].

1.3 Contributions

This work introduces a system of **typed syntax macros (TSMs)** 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.

Client programmers apply TSMs to *generalized literal forms*. For example, in Figure 1.4 we apply a TSM named `$html` to a generalized literal form delimited by backticks. TSM names are prefixed by `$` to clearly distinguish TSM application from function application. The semantics delegates control over the parsing and expansion of each literal body to the applied TSM during a semantic phase called *typed expansion*, which generalizes the usual typing phase.

```
$html `<p>Hello, {[join ($str ' ') ($strlist [first, last])]}</p>`
```

Figure 1.4: An example of a TSM being applied to a generalized literal form. The literal body, in green, is initially left unparsed according to the language's context-free syntax.

Generalized literal forms subsume a variety of common syntactic forms because the context-free syntax of the language only defines which outer delimiters are available. *Literal bodies* (in green in Figure 1.4) are otherwise syntactically unconstrained and left unparsed. For example, the `$html` TSM is free to use an Ur/Web-inspired HTML syntax (compare Figure 1.4 to Figure 1.2.) This choice is not imposed by the language definition. Generalized literal forms have no TSM-independent meaning.

The primary technical challenge has to do with the fact that the applied TSM needs to be able to parse terms out of the literal body for inclusion in the expansion. We refer to these as *spliced terms*. For example, Figure 1.5 reveals the locations of the spliced expressions in Figure 1.4 by coloring them black. We have designed our system so that a figure like this, which presents a *segmentation* of each literal body into spliced terms (in black) and characters parsed in some other way by the applied TSM (in color), can always be automatically generated no matter how each applied TSM has been implemented.

```
$html ‘<p>Hello, {[join ($str ’ ’) ($strlist [first, last])]}</p>‘
```

Figure 1.5: The segmentation of the example from Figure 1.4

Notice that both arguments to `join` are themselves of TSM application form – the TSMs named `$str` and `$strlist` are applied to generalized literal forms delimited by quotation marks and square brackets, respectively. The bracket-delimited literal form, in turn, contains two spliced expressions of variable form – `first` and `last`.

TSMs come equipped with useful principles of syntactic abstraction. We will more precisely characterize these abstract reasoning principles as we proceed. For now, to develop some intuitions, consider Figure 1.6, which uses TSMs to express the "unreasonable" example from Figure 1.3.

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

Figure 1.6: TSMs make examples like the one from Figure 1.3 more reasonable.

Without examining the expansion of Line 3, we can reason as follows:

1. **(Responsibility)** The applied TSM, `$kquery`, is solely responsible for typed expansion of the literal body.
2. **(Segmentation)** By examining the segmentation, we know that the two instances of `x` on Line 3 are parsed as spliced expressions, whereas `R` is parsed in some other way peculiar to this form.
3. **(Capture)** The system prevents capture, so the spliced expression `x` must refer to the binding of `x` on Line 2 – it cannot capture an unseen binding introduced in the expansion of Line 3.
4. **(Context Dependence)** The system enforces context independence, so the expansion of Line 3 cannot rely on the fact that, for example, `w` is in scope.

5. **(Typing)** An explicit type annotation on the definition of `$kquery` determines the type that every expansion it generates will have. We will see an example of a TSM definition in Chapter 3.

   Moreover, each segment in the segmentation also comes paired with the type it is expected to have. This information is usually not necessary to reason about typing, but it can be conveyed to the programmer upon request by the program editor if desired.

1.3.1 Outline

After introducing necessary background material and summarizing the related work in greater detail in Chapter 2, we formally introduce TSMs in Chapter 3 by integrating them into a simple language of expressions and types. The introductory examples above can be expressed using the language introduced in Chapter 3.

In Chapter 4, we add structural pattern matching to the language of Chapter 3 and introduce *pattern TSMs*, i.e. TSMs that generate patterns rather than expressions.

In Chapter 5, we equip the language of Chapter 4 with type functions and an ML-style module system. We then introduce *parametric TSMs*, i.e. TSMs that take type and module parameters. Parameters serve two purposes:

1. They enable TSMs that operate not just at a single type, but over a type- and module-parameterized family of types. For example, rather than defining a TSM `$strlist` for string lists and another TSM `$intlist` for integer lists, we can define a single parametric TSM `$list` that operates uniformly across the type-parameterized family of list types.
2. They allow the expansions that TSMs generate to refer to application site bindings in a context independent manner.

We also demonstrate support for partial parameter application in TSM abbreviations, which decreases the syntactic cost of this explicit parameter passing style. Figure 1.7 demonstrates all of these features.

```
let syntax $strlist = $list string in
$html ‘<p>Hello, {[join ($str ’ ’) ($strlist [first, last])]}</p>‘
```

Figure 1.7: The example from Figure 1.5 expressed using parametric TSMs

In these first chapters, we assume for the sake of technical simplicity that each TSM definition is self-contained, needing no access to libraries or to other TSMs. This is an impractical assumption in practice. We relax this assumption in Chapter 6, introducing a *static environment* shared between TSM definitions. We also give examples of TSMs that are useful for defining other TSMs, e.g. TSMs that implement parser generators and quasiquotation.

In Chapter 7, we develop a mechanism of *TSM implicits* that allows library clients to contextually designate, for any type, a privileged TSM at that type. The semantics applies this privileged TSM implicitly to unadorned literal forms that appear where a term of the associated type is expected. For example, if we designate `$str` as the privileged TSM at the `string` type and `$strlist` as the privileged TSM at the `list(string)` type, we can express the example from Figure 1.5 instead as shown in Figure 1.8 (assuming `join` has type `string -> list(string) -> string`.)

```
$html‘<p>Hello, {[join ’ ’ [first, last]]}</p>‘
```

Figure 1.8: The example from Figure 1.5 drawn to take advantage of TSM implicits

This approach is competitive in cost with library-specific syntax dialects (e.g. compare Figure 1.8 to Figure 1.2), while maintaining the abstract reasoning principles characteristic of our approach. To further demonstrate the favorable economics of this approach, Figure 1.9 gives an example of a function that produces a value of type `html`. The body of this function assumes implicit TSM designations at seven different types (the unspliced segments are typeset in a color corresponding to the type that the enclosing literal form is being checked against.) This collection of TSMs, together with the mechanism for applying them implicitly, obviates the need for a web-programming-specific syntax dialect of our language like Ur/Web.

```
fun resultsFor(searchQuery : string, page : int) : html =>
  let imageBase : url = ‘images.example.com‘ in
  let bgImage : url = ‘$imageBase$/background.png‘ in
  ‘<html>
  <head>
    <title>Search Results</title>
    <style>{
      body { background-image: url({bgImage}) }
      .search { background-color: {darken(‘#aabbcc‘, ‘10%‘)} }
    }</style>
  </head><body>
    <h1>Results for {[searchQuery]}</h1>
    <div class="search">
      Search again: {searchBox "Go!"}
    </div>
    {formatResults (db,
       ‘SELECT * FROM products WHERE {searchQuery} in title‘,
       10, page)}
  </body>
  </html>‘
```

Figure 1.9: A non-trivial example demonstrating implicit TSM application at seven different types: `url`, `html`, `css`, `color`, `percentage`, `string` and `sql`

We conclude in Chapter 8 with a discussion of the present limitations of TSMs, and outline various directions for future work.

1.3.2 Thesis Statement

In summary, this work defends the following statement:

> A programming language (in the ML tradition) can give library providers the ability to programmatically control the parsing and expansion of expressions and patterns of generalized literal form such that clients can reason abstractly about responsibility, segmentation, types and binding.

1.4 VerseML

The code examples in this document are written in a new full-scale functional language called VerseML.[6] VerseML is the language of Chapter 7 extended with some additional conveniences that are commonly found in other functional languages (in particular, in the ML family of languages) and, notionally, orthogonal to TSMs (e.g. higher-rank polymorphism [39], signature abbreviations, and syntactic sugar that is not library-specific, e.g. for curried functions.) We will not formally define these features mainly to avoid unnecessarily complicating our presentation with details that are not essential to the ideas introduced herein. As such, all examples written in VerseML should be understood to be informal motivating material for the subsequent formal material.

1.5 Disclaimers

Before we continue, it may be prudent to explicitly acknowledge that eliminating the need for syntax dialects would indeed be asking for too much: certain syntax design decisions are fundamentally incompatible with others or require coordination across a language design. We aim only to diminish the need for syntax dialects by finding a reasonable "sweet spot" in the design space, not to give control over all design decisions to library providers.

It may also be prudent to explicitly acknowledge that library providers could use TSMs to define syntactic forms that are "in poor taste." In practice, programmers should defer to established community guidelines before defining their own TSMs (following the example of languages that support operator overloading or *ad hoc* polymorphism using type classes [38, 58], which also have some potential for "abuse" or "overuse".) The majority of programmers should very rarely need to define a TSM on their own. The reasoning principles that we will develop ensure that even poorly designed TSMs cannot prevent clients from reasoning abstractly about types and binding.

[6] We distinguish VerseML from Wyvern, which is the language described in our prior publications about some of the work that we will describe, because Wyvern is a group effort evolving independently.

Chapter 2

Background

> *The recent development of programming languages suggests that the simultaneous achievement of simplicity and generality in language design is a serious unsolved problem.*
>
> John Reynolds (1970) [103]

2.1 Preliminaries

This work is rooted in the tradition of full-scale functional languages like Standard ML, OCaml and Haskell (as might have been obvious from Chapter 1.) Familiarity with basic concepts in these languages, e.g. variables, types, polymorphic and recursive functions, tuples, records, recursive datatypes and structural pattern matching, is assumed throughout this work. Readers who are not familiar with these concepts are encouraged to consult the early chapters of an introductory text like Harper's *Programming in Standard ML* [59] (a working draft can be found online.) We discuss integrating TSMs into languages from other design traditions in Sec. 8.2.3.

In Chapter 5 and onward, as well as in some of the motivating examples below, we also assume basic familiarity with ML-style module systems. Readers with experience in a language without such a module system (e.g. Haskell) are also advised to consult the relevant chapters in *Programming in Standard ML* [59] as needed. We distinguish *modules*, which are language constructs, from *libraries*, which are extralinguistic packaging constructs managed by some implementation-defined compilation manager (e.g. `CM`, distributed with Standard ML of New Jersey (SML/NJ) [17].) A library can export modules, signatures and TSM definitions. We return to this distinction in Chapter 6.

The formal systems that we will consider are defined within the metatheoretic framework of type theory. More specifically, we will assume that abstract binding trees (ABTs, which enrich abstract syntax trees with the notions of binding and scope, as discussed in Chapter 1), renaming, alpha-equivalence, substitution, structural induction and rule induction are defined as described in Harper's *Practical Foundations for Programming Languages, Second Edition* (*PFPL*) [61]. Familiarity with other formal accounts of type systems, e.g. Pierce's *Types and Programming Languages* (*TAPL*) [99], should also suffice.

2.2 Cognitive Cost

In the present inquiry, the idea is to adopt a much wider conception of formal languages so as to investigate more broadly what exactly is going on when a reasoner puts these tools to use.

Catarina Dutilh Novaes
Formal Languages in Logic: A Philosophical and Cognitive Analysis [92]

Central to our motivations is the notion that different drawings of a formal structure can and should be distinguished on the basis of the *cognitive costs* that humans incur as they interact with them.

The broad notion of cognitive cost must ultimately be understood intuitively, relating as it does to the complexities of the human mind. Cognitive cost is also fundamentally a *subjective* and *situational* notion. As such, researchers cannot develop a truly comprehensive formal framework capable of settling questions of cognitive cost.[1] However, there are several situationally useful frameworks worth briefly reviewing [20].

One useful quantitative framework reduces cognitive cost to *syntactic cost*, which is measured by counting characters (or glyphs, more generally.) This is often a satisfying proxy for cognitive cost, in that smaller drawings are often easier to comprehend and produce. For example, the drawing `[x, y, z]` has lower syntactic cost than its desugaring, as discussed in the previous chapter. There is a limit to this approximation, of course. For example, one might argue that the drawings involving the syntax of K, like the drawing from Figure 1.3, have high cognitive cost, despite their low syntactic cost, until one is experienced with the syntax of K. In other words, the relationship between syntactic cost and cognitive cost depends on the subject's progression along some *learning curve*.

A related quantity of interest to human programmers is *edit cost*, measured relative to a program editor as the minimum number of primitive edit actions that must be performed to produce a drawing. For example, when using a text editor (as most professional programmers today do), drawings in textual form typically have lower edit cost, as measured by the minimum number of keystrokes necessary to produce the drawing, than those in operational or stylized forms (indeed, some drawings in stylized form can be understood to have infinite text edit cost.) Edit cost can be modeled using, for example, *keystroke-level models* (KLMs) as described by Card, Moran and Newell [24].

One can also analyze cognitive cost using disciplined qualitative methods. Green's *Cognitive Dimensions of Notations* [55, 56] and Pane and Myers' *Usability Issues* [98] (both of which synthesized much of the earlier work in the area) are highly cited heuristic frameworks. For example, Green's cognitive dimensions framework gives us a common vocabulary for comparing the derived list forms described in Chapter 1 to the primitive list forms. In particular, the derived list forms *map more closely* to other notations used for sequences of elements (e.g. in typeset mathematics, or on a physical notepad) than the

[1] The fact that cognitive cost cannot be comprehensively characterized seems itself to create a cognitive hazard, in that those of us who favor comprehensive formal frameworks sometimes devalue or dismiss concerns related to cognitive cost, or consider them in an overly *ad hoc* manner. This tendency must be resisted if programming language design is to progress as a human-oriented design discipline.

primitive list forms. They also make the elements of the list more clearly *visible*, in that the identifier `Cons` is not interspersed throughout the term, and they have lower *viscosity* because adding a new item to the middle of a list drawn in derived form requires only a local edit, whereas for a list constructed by applying list constructors in prefix position, one needs also to add a closing parenthesis to the end of the term. (Infix operators for lists, discussed in Sec. 2.4.3, also have low viscosity.)

Finally, one might consider cognitive cost comparatively using quantitative empirical methods, e.g. by conducting randomized control trials to compare forms with respect to task completion time or error rate (for satisfyingly representative tasks.) Stefik et al. have performed many such studies, mainly on novice programmers (these are summarized, along with other such studies, in [116].) Kaijanaho provides another review of evidence-based language design methodologies [74].

Our goal in this work is to provide a means by which library providers can introduce alternative syntactic forms of their own design. We leave it up to each library provider to establish the cognitive costs associated with the alternative forms that they introduce, according to whichever operationalization of the concept that they favor. For the examples in this document, we will mainly utilize syntactic cost, because claims about syntactic cost can be evaluated quantitatively. In a few cases, we also make heuristic arguments.

We claim also that the abstract reasoning principles that TSMs come equipped with serve to limit cognitive costs that a client programmer that encounters an unfamiliar form would otherwise incur when attempting to reason about types and binding. This claim follows from the intuitive assumption that examining only type annotations is less costly than examining the full expansion of an unexpanded term and the logic that produced that expansion.

2.3 Motivating Definitions

In this section, we give a number of VerseML definitions that will serve as the basis for many subsequent examples. This section also serves as an introduction to the textual syntax and semantics of VerseML.

2.3.1 Lists

The Standard ML Basis Library (i.e. the standard library) defines list types as follows:

```
datatype 'a list = nil | op:: of 'a * 'a list
```

This datatype declaration generates:

- a type function `list` that takes one type parameter;
- the value constructors `nil : 'a list` and `op:: : 'a * 'a list -> 'a list`; and
- the corresponding list pattern constructors `nil` and `op::`.

We will return to the significance of the identifier `op::` in Sec. 2.4.3 below.

VerseML does not support SML-style datatype declarations directly. Instead, type functions, recursive types, sum types, product types, value constructors, pattern constructors and type generativity arise through orthogonal mechanisms, as in foundational accounts of these concepts (e.g. *PFPL* [61].) This is mainly for pedagogical purposes – it will take until Chapter 5 to build up all of the machinery that would be necessary to integrate TSMs into a language with SML-style datatype declarations. By exposing more granular primitives, we can define sub-languages of VerseML in Chapter 3 and Chapter 4 that communicate certain fundamental ideas more clearly and generally.

With that in mind, the family of list types are defined in VerseML as follows:

```
type list('a) = rec(self => Nil + Cons of 'a * self)
```

Here, `list` is a type function binding its type parameter to the type variable `'a`. We apply parameters in post-fix position (rather than in prefix position, as in SML.) For example, the type of integer lists is `list(int)`. This is equivalent, by substitution of `int` for `'a` on the right side of the definition above, to the following *recursive type*:

```
rec(self => Nil + Cons of int * self)
```

The values of a recursive type `T` are **`fold`**`(e)`, where `e` is a value of the *unrolling* of `T`. The unrolling of a recursive type is determined by substituting the recursive type itself for the self reference in its type body. For example, the unrolling of `list(int)` is equivalent, by substitution of `list(int)` for `self`, to the following *labeled sum type*:

```
Nil + Cons of int * list(int)
```

The values of a labeled sum type `T` are injections **`inj`**`[Lbl](e)`, where `Lbl` is a label specified by one of the classes specified by `T` and `e` is a value of the corresponding type. The labeled sum type above specifies two classes:

1. One class, labeled `Nil`, takes values of `unit` type (we can omit **`of`** `unit`.) The only value of `unit` type is the trivial value `()`.
2. The other class, labeled `Cons`, takes values of the *product type* `int * list(int)`, the values of which are tuples.

Let us now define two example values of type `list(int)`:

```
val nil_int : list(int) = fold(inj[Nil] ())
val one_int : list(int) = fold(inj[Cons] (1, nil_int))
```

Here, `nil_int` is the empty list and `one_int` is a list containing a single integer, `1`.

One way to lower syntactic cost is to define the following polymorphic values, called the *list value constructors*, which abstract away the necessary folds and injections:

```
val Nil : list('a) = fold(inj[Nil] ())
fun Cons(x : 'a * list('a)) : list('a) => fold(inj[Cons] x)
```

In fact, VerseML generates constructors like these automatically.[2] Using these list value constructors, we can equivalently express the values above as follows:

```
val nil_int : list(int) = Nil
val one_int = Cons (1, Nil)
```

In SML, constructors like these are the only means by which a value of a datatype can be introduced – folding and injection operators are not exposed directly to programmers.

[2]A more general mechanism that allows values to be generated from type definitions is beyond the scope of our work on TSMs.

As such, it is not possible to construct a value of a type like `list(int)` in a context-independent manner, i.e. in contexts where the value constructors have been shadowed or are not bound. This will become relevant in the next section and in Chapter 3.

Values of recursive type, labeled sum type and product type are deconstructed by pattern matching. For example, we can write the polymorphic map function, which constructs a list by applying a given function to each item in a given list, as follows:

```
fun map (f : 'a -> 'b) (xs : list('a)) : list('b) =>
  match xs with
  | fold(inj[Nil] ()) => Nil
  | fold(inj[Cons] (y, ys)) => Cons (f y, map f ys)
  end
```

The primitive pattern forms above are drawn like the corresponding primitive value forms (though it is important to keep in mind that the syntactic overlap is superficial – patterns and expressions are distinct sorts of trees.) To lower syntactic cost, VerseML automatically inserts folds, injections and trivial arguments into patterns of constructor form, i.e. those of the form `Lbl` and `Lbl p` where `Lbl` is a capitalized label and `p` is another pattern:[3]

```
fun map (f : 'a -> 'b) (xs : list('a)) : list('b) =>
  match xs with
  | Nil => Nil
  | Cons (y, ys) => Cons (f y, map f ys)
  end
```

We group the type and value definitions above, as well as some other standard utility functions like `append`, into a *module* `List : LIST`, where `LIST` is the *signature* defined in Figure 2.1. These definitions are not privileged in any way by the language definition. In particular, there are no list-specific derived forms built in to the textual syntax of VerseML. We will show how TSMs allow programmers to achieve a similar syntax for lists over the next several chapters.

```
signature LIST =
sig
  type list('a) = rec(self => Nil + Cons of 'a * self)
  val Nil : list('a)
  val Cons : 'a * list('a) -> list('a)
  val map : ('a -> 'b) -> list('a) -> list('b)
  val append : list('a) -> list('a) -> list('a)
  (* ... *)
end
```

Figure 2.1: Definition of the `LIST` signature

[3]Pattern TSMs, introduced in Chapter 4, could be used to manually achieve a similar syntax for any particular type, or in Chapter 5, across a particular family of types, but because this syntactic sugar is useful for all recursive labeled sum types, we build it primitively into VerseML.

2.3.2 Regular Expressions

A regular expression, or *regex*, is a description of a *regular language* [123]. Regexes arise with some frequency in fields like natural language processing and bioinformatics.

Recursive Sums One way to encode regular expressions in VerseML is as values of the recursive labeled sum type abbreviated `rx` in Figure 2.2.

```
type rx = rec(rx => Empty + Str of string + Seq of rx * rx +
                    Or of rx * rx + Star of rx)
```

Figure 2.2: Definition of the recursive labeled sum type `rx`

Assuming the automatically generated value constructors as in Sec. 2.3.1, we can construct a regex that matches the strings "A", "T", "G" or "C" (i.e. DNA bases) as follows:

```
Or(Str "A", Or(Str "T", Or(Str "G", Str "C")))
```

Given a value of type `rx`, we can deconstruct it by pattern matching, again as in Sec. 2.3.1. For example, the function `is_dna_rx` defined in Figure 2.3 detects regular expressions that match DNA sequences.

```
fun is_dna_rx(r : rx) : boolean =>
  match r with
  | Str "A" => True
  | Str "T" => True
  | Str "G" => True
  | Str "C" => True
  | Seq (r1, r2) => (is_dna_rx r1) andalso (is_dna_rx r2)
  | Or  (r1, r2) => (is_dna_rx r1) andalso (is_dna_rx r2)
  | Star(r') => is_dna_rx r'
  | _ => False
  end
```

Figure 2.3: Pattern matching over regexes in VerseML

Abstract Types Encoding regexes as values of type `rx` is straightforward, but there are reasons why one might not wish to expose this encoding to clients directly.

First, regexes are usually identified up to their reduction to a normal form. For example, `Seq(Empty, Str "A")` has normal form `Str("A")`. It can be useful for regexes with the same normal form to be indistinguishable from the perspective of client code. (The details of regex normalization are not important for our purposes, so we omit them.)

Second, it can be useful for performance reasons to maintain additional data alongside each regex (e.g. a corresponding finite automaton.) In fact, there may be many ways to represent regexes, each with different performance trade-offs, so we would like to provide a choice of representations behind a common interface.

To achieve these goals, we turn to the VerseML module system, which is based directly on the SML module system [37, 89] (which originates in early work by MacQueen [83].) In particular, let us define the signature abbreviated `RX` in Figure 2.4.

```
(* abstract regex unfoldings *)
type u('a) = UEmpty + UStr of string + USeq of 'a * 'a +
             UOr of 'a * 'a + UStar of 'a

signature RX =
sig
  type t (* abstract *)

  (* constructors *)
  val Empty : t
  val Str : string -> t
  val Seq : t * t -> t
  val Or : t * t -> t
  val Star : t -> t

  (* produces the normal unfolding *)
  val unfold_norm : t -> u(t)
end

module R1 : RX = struct (* ... *) end
module R2 : RX = struct (* ... *) end
```

Figure 2.4: The `RX` signature and two example implementations

The clients of any module `R` that has been sealed by `RX`, e.g. `R1` or `R2` in Figure 2.4, manipulate regexes as values of type `R.t` using the interface specified by `RX`. For example, a client can construct a regex matching DNA bases by projecting the value constructors out of `R` and applying them as follows:

```
R.Or(R.Str "A", R.Or(R.Str "T", R.Or (R.Str "G", R.Str "C")))
```

Because the identity of the representation type `R.t` is held abstract by the signature, the only way for a client to construct a value of this type is through the values that `RX` specifies (i.e. we have defined an *abstract data type (ADT)* [80].) Consequently, representation invariants need only be established locally within each module.

Similarly, clients cannot interrogate the structure of a value `r : R.t` directly. Instead, the signature specifies a function `R.unfold_norm` that produces the *normal unfolding* of a given regex, i.e. a value of type `u(R.t)` that exposes only the outermost form of the regex in normal form (this normal form invariant is specified only as an unenforced side condition that implementations are expected to obey, as is common practice in languages like ML.) Clients can pattern match over the normal unfolding in the familiar manner, as shown in Figure 2.5.

The normal unfolding suffices in situations where a client needs to examine only the outermost structure of a regex. However, in general, a client may want to pattern match more deeply into a regex. There are various ways to approach this problem.

```
fun is_dna_rx'(r : R.t) : boolean =>
  match R.unfold_norm r with
  | UStr "A" => True
  | UStr "T" => True
  | UStr "G" => True
  | UStr "C" => True
  | USeq (r1, r2) => (is_dna_rx' r1) andalso (is_dna_rx' r2)
  | UOr (r1, r2) => (is_dna_rx' r1) andalso (is_dna_rx' r2)
  | UStar r' => is_dna_rx' r'
  | _ => False
  end
```

Figure 2.5: Pattern matching over normal unfoldings of regexes

```
functor RXUtil(R : RX) =
struct
  fun unfold_norm2(r : R.t) : u(u(R.t)) => (* ... *)

  fun view(r : R.t) : rx =>
    match R.unfold_norm r with
    | UEmpty => Empty
    | UStr s => Str s
    | USeq (r1, r2) => Seq (view r1, view r2)
    | UOr (r1, r2) => Or (view r1, view r2)
    | UStar r => Star (view r)
    end

  (* ... *)
end
```

Figure 2.6: The definition of `RXUtil`

One approach is to define auxiliary functions that construct n-deep unfoldings of `r`, where n is the deepest level at which the client wishes to expose the normal structure of the regex. For example, it is easy to define a function `unfold_norm2 : R.t -> u(u(R.t))` in terms of `R.unfold_norm` that allows pattern matching to depth 2.[4]

Another approach is to *completely unfold* a value of type `t` by applying a function `view : R.t -> rx` that recursively applies `R.unfold_norm` to exhaustion. The type `rx` was defined in Figure 2.2. Computing the complete unfolding (also called the *view*) can have higher dynamic cost than computing an incomplete unfolding of appropriate depth, but it is also a simpler approach (i.e. lower cognitive cost can justify higher dynamic cost.)

Typically, utility functions like `unfold_norm2` and `view` are defined in a *functor* (i.e. a function at the level of modules) like `RXUtil` in Figure 2.6, so that they need only be defined once, rather than separately for each module `R : RX`. The client can instantiate the functor by applying it to their choice of module as follows:

```
module RU = RXUtil(R)
```

[4]Defining an unfolding *generic* in n is a more subtle problem that is beyond the scope of this work.

2.4 Existing Approaches

The definitions in the previous section adequately encode the semantics of lists and regular expressions, but they are not particularly convenient. Our task in this section is to consider various mechanisms of syntactic control, i.e. mechanisms that can be deployed to help to decrease the syntactic cost of expressions and patterns involving these constructs (without changing their meaning.)

We begin in Sec. 2.4.1 by considering standard abstraction mechanisms available in languages like ML. We then consider a system of dynamic quotation parsing available in some dialects of ML in Sec. 2.4.2.

These methods give library providers only limited control over form and operate at "run-time." To gain more precise control over form at "compile-time", a library provider, or another interested party, can define a "library-specific" syntax dialect using a *syntax definition system*. The next several sections consider various syntax definition systems:

- In Sec. 2.4.3, we consider infix operator definition systems.
- In Sec. 2.4.4, we consider somewhat more expressive mixfix systems.
- In Sec. 2.4.5, we consider grammar-based syntax definition systems.
- In Sec. 2.4.6, we consider parser combinator systems.

The systems in Sec. 2.4.5 and Sec. 2.4.6 give essentially complete control over form to their users. We give examples of dialects that can be constructed using these systems in Sec. 2.4.7. Then, in Sec. 2.4.8, we discuss the difficulties that programmers can expect to encounter if they use these systems when programming in the large (as a follow-up to what was discussed in Section 1.2.1.)

An alternative approach is to leave the syntax of the language fixed but allow programmers to contextually repurpose existing forms using a *term rewriting system*. We consider non-local term rewriting systems in Sec. 2.4.9 and local term rewriting systems, which are also known as *macro systems*, in Sec. 2.4.10.

2.4.1 Standard Abstraction Mechanisms

The simplest way to decrease syntactic cost is to capture idioms using the standard abstraction mechanisms of our language, e.g. functions and modules.

We already saw examples of this approach in the previous section. For example, we defined the list value constructors, which capture the idioms of list construction. Such definitions are common enough that VerseML generates them automatically. We also defined a utility functor for regexes, `RXUtil`, in Figure 2.6. As more idioms involving regexes arise, the library provider can capture them by adding additional definitions to this functor. For example, the library provider might add the definition of a value that matches single digits to `RXUtil` as follows:

```
val digit = R.Or(R.Str "0", R.Or(R.Str "1", ...))
```

Similarly, the library provider might define a function `repeat : R.t -> int -> R.t` that constructs a regex by sequentially repeating the given regex a given number of times (not shown.) Using these definitions, a client can define a regex that matches U.S. social security numbers (SSNs) as follows:

```
val dash = R.Str "-"
val repeat_d = RU.repeat RU.digit
val ssn = R.Seq(repeat_d 3, R.Seq(dash, R.Seq(repeat_d 2,
                R.Seq(dash, repeat_d 4))))
```

The syntactic cost of this program fragment is lower than the syntactic cost of the equivalent program fragment that applies the regex value constructors directly.

One limitation of this approach is that there is no standard way to capture idioms at the level of patterns. Pattern synonyms have been informally explored in some languages, e.g. in an experimental extension of Haskell implemented by GHC [1] and in the Ωmega language [110], but these are limited in that arbitrary computations cannot be performed.

Another limitation is that this approach does not give library providers control over form. For example, we cannot "approximate" SML-style derived list forms using only auxiliary values like those above. Similarly, consider the textual syntax for regexes defined in the POSIX standard [7]. Under this syntax, the regex that matches DNA bases is drawn as follows:

```
A|T|G|C
```

Similarly, the regex that matches SSNs is drawn:

```
\d\d\d-\d\d-\d\d\d\d
```

or

```
\d{3}-\d{2}-\d{4}
```

These drawings have substantially lower syntactic cost than the drawings of the corresponding VerseML encodings shown above. Data suggests that most professional programmers are familiar with POSIX regex forms [95]. These programmers would likely agree that the POSIX forms have lower cognitive cost as well.

Dynamic String Parsing

We might attempt to approximate the POSIX standard regex syntax by defining a function `parse : string -> R.t` in `RXUtil` that parses a VerseML string representation of a POSIX regex form, producing a regular expression value or raising an exception if the input is malformed with respect to the POSIX specification. Given this function, a client could construct the regex matching DNA bases as follows:

```
RU.parse "A|T|G|C"
```

This approach, which we refer to as *dynamic string parsing*, has several limitations:

1. First, there are syntactic conflicts between standard string escape sequences and standard regex escape sequences. For example, the following is not a well-formed drawing according to the textual syntax of SML (and many other languages):

   ```
   val ssn = RU.parse "\d\d\d-\d\d-\d\d\d\d" (* ERROR *)
   ```

In practice, most parsers report an error message like the following:[5]

```
error: illegal escape character
```

In a small lab study, we observed that even experienced programmers made this class of mistake and could not quickly diagnose the problem and determine a workaround if they had not used a regex library recently [95].

The workaround – escaping all backslashes – nearly doubles syntactic cost here:

```
val ssn = RU.parse "\\d\\d\\d-\\d\\d-\\d\\d\\d\\d"
```

Some languages build in alternative "raw" string forms that leave escape sequences uninterpreted. For example, OCaml supports alternative string literals delimited by matching marked curly braces, e.g.

```
val ssn = RU.parse {rx|\d\d\d-\d\d-\d\d\d\d|rx}
```

2. The next limitation is that dynamic string parsing does not capture the idioms of compositional regex construction. For example, the function `lookup_rx` in Figure 2.7 constructs a regex from the given string and another regex. We cannot apply `RU.parse` to redraw this function equivalently, but at lower syntactic cost.

```
fun lookup_rx(name : string) =>
  R.Seq(R.Str name, R.Seq(R.Str ": ", ssn))
```

Figure 2.7: Compositional construction of a regex

We will describe derived forms that do capture the idioms of compositional regex construction in Sec. 2.4.5 (in particular, we will compare Figure 2.7 to 2.9.)

Dynamic string parsing cannot capture the idioms of list construction for the same reason – list expressions can contain sub-expressions.

3. Using strings to introduce regexes also creates a *cognitive hazard* for programmers who are coincidentally working with other data of type `string`. For example, consider the following naïvely "more readable definition of `lookup_rx`", where the infix operator `^` means string concatenation:

```
fun lookup_rx_insecure(name : string) =>
  RU.parse (name ^ {rx|: \d\d\d-\d\d-\d\d\d\d|rx})
```

or equivalently, given the regex `ssn` as above and an auxiliary function `RU.to_string` that can compute the string representation of a given regex:

```
fun lookup_rx_insecure(name : string) =>
  RU.parse (name ^ ": " ^ (RU.to_string ssn))
```

Both `lookup_rx` and `lookup_rx_insecure` have the same type, `string -> R.t`, and behave identically at many inputs, particularly the "typical" inputs (i.e. alphabetic strings.) It is only when `lookup_rx_insecure` is applied to a string that parses as

[5]This is the error message that `javac` produces. When compiling an analagous expression using SML of New Jersey (SML/NJ), we encounter a more confusing error message: `Error: unclosed string`.

a regex that matches *other* strings that it behaves incorrectly (i.e. differently from `lookup_rx`.)

In applications that query sensitive data, mistakes like this lead to *injection attacks*, which are among the most common and catastrophic security threats today [10].

This problem is fundamentally attributable to the programmer making a mistake in a misguided effort to decrease syntactic cost. However, the availability of a better approach for decreasing syntactic cost would help make this class of mistakes less common [22].

4. The final problem is that regex parsing does not occur until the call to `RU.parse` is dynamically evaluated. For example, the malformed regex form in the program fragment below will only trigger an exception when this expression is evaluated during the full moon:

```
match moon_phase with
Full => RU.parse "(GC" | _ => (* ... *)
end
```

Malformed string encodings of regexes can sometimes be discovered by testing, though empirical data gathered from large open source projects suggests that many malformed regexes remain undetected by test suites "in the wild" [115].

One workaround is for the programmer to lift all such calls where the argument is a string literal out to the top level of the program, so that the exception is raised every time the program is evaluated. There is a cognitive penalty associated with moving the description of a regex away from its use site (but for statically determined regexes, this might be an acceptable trade-off.) For regexes constructed compositionally, this may not be possible.

Another approach is to perform a static analysis that attempts to discover malformed statically determined regexes wherever they appear [115].

5. Finally, to reiterate, this approach is not suitable for abbreviating patterns.

Difficulties like these arise whenever a programmer attempts to deploy dynamic string parsing as a solution to the problem of high syntactic cost. (There are, of course, legitimate applications of dynamic string parsing that are not motivated by the desire to decrease syntactic cost, e.g. when parsing string encodings of regexes received as dynamic input to the program.)

2.4.2 Dynamic Quotation Parsing

Some syntax dialects of ML, e.g. a syntax dialect that can be activated by toggling a compiler flag in SML/NJ [6, 113], define *quotation literals*, which are derived forms for expressions of type `'a frag list` where `'a frag` is defined as follows:

```
datatype 'a frag = QUOTE of string | ANTIQUOTE of 'a
```

Quotation literals are delimited by backticks, e.g. `` `A|T|G|C` `` is the same as writing `[QUOTE "A|T|G|C"]`. Expressions of variable or parenthesized form that appear prefixed by a caret in the body of a quotation literal are parsed out and appear wrapped in the `ANTIQUOTE` constructor, e.g. `` `GC^(dna_rx)GC` `` is the same as writing:

```
[QUOTE "GC", ANTIQUOTE dna_rx, QUOTE "GC"]
```

Unlike dynamic string parsing, *dynamic quotation parsing* allows library providers to capture idioms involving subexpressions. For example:

- The regex library provider can define a function `qparse : R.t frag list -> R.t` in `RXUtil` that parses the given fragment list according to the POSIX standard extended to support antiquotation, producing a regex value or raising an exception if the fragment list cannot be parsed. Appyling this function to the examples above produces the corresponding regex values at lower syntactic cost:

  ```
  val dna = RU.qparse `A|T|G|C`
  val bisI = RU.qparse `GC^(dna_rx)GC`
  ```

- The list library provider can also define a function `qparse : 'a frag list -> 'a list` in the `List` module that constructs a list from a quoted list:

  ```
  List.qparse `[^(x + y), ^y, ^z]`
  ```

There remain some problems with dynamic quotation parsing:

1. The library provider cannot specify alternative outer delimiters or antiquotation delimiters – backticks and the caret, respectively, are the only choices in SML/NJ. This is problematic for regexes, for example, because the caret has a different meaning in the POSIX standard.

2. Another problem is that all antiquoted values within a quotation literal must be of the same type. If, for example, we sought to support both spliced regexes and spliced strings in quoted regexes, we would need to define an auxiliary sum type in `RXUtil` and the client would need to wrap each antiquoted expression with a call to the corresponding constructor to mark its type. For example, `lookup_rx` would be drawn as follows (assuming suitable definitions of `RU.QS` and `RU.QR`, not shown):

   ```
   fun lookup_rx(string : name) =>
     RU.qparse' `^(RU.QS name): ^(RU.QR reading)`
   ```

   Similarly, if we sought to support quoted lists where the tail is explicitly given by the client (following OCaml's revised syntax [79]), clients would need to apply marking constructors to each antiquoted expression:

   ```
   List.qparse `[^(List.V x), ^(List.V y) :: ^(List.VS zs)]`
   ```

   Marking constructors increase syntactic cost (rather substantially in such examples.)

3. As with dynamic string parsing, parsing occurs dynamically. We cannot use the trick of lifting all calls to `qparse` to the top level because the arguments are not closed string literals. At best, we can lift these calls out as far as the binding structure allows, i.e. into the earliest possible "dynamic phase." Parse errors are detected only when this phase is entered, and the dynamic cost of parsing is incurred each time this phase is entered. For example, `List.qparse` is called n times below, where n is the length of `input`:

```
List.map (fn x => List.qparse `[^x, ^(2 * x)]`) input
```

One way to detect parse errors early and reduce the dynamic cost of parsing is to use a system of *staged partial evaluation* [71]. For example, if we integrated Davies' temporal logic based approach into our language [34], we could rewrite the list example above as follows:

```
List.map (fn x => prev (List.sqparse
  `[^(next x), ^(next (2 * x))]`)) input
```

Here, the operator **prev** causes the call to `List.sqparse` to be evaluated in the previous stage. `List.sqparse` differs from `List.qparse` in that the antiquoted values in the input must be encapsulated expressions from the next stage, indicated by the **next** operator. The return value is also an encapsulated expression from the next stage. By composing this value with **prev**, we achieve the desired staging. Other systems, e.g. MetaML [109] and MacroML [53], provide similar staging primitives.

The main problem with this approach is that it incurs substantial annotation overhead. Here, the staged call to `List.sqparse` has higher syntactic cost than if we had simply manually applied `Nil` and `Cons`. This problem is compounded if marking constructors like those described above are needed.

4. Finally, quotation parsing, like the other approaches considered so far, helps only with the problem of abbreviating expressions. It provides no solution to the problem of abbreviating patterns (because parse functions compute values, not patterns.)

Due to these problems, VerseML does not build in quotation literals.[6]

2.4.3 Fixity Directives

We will now consider various syntax definition systems.

The simplest syntax definition systems allow programmers to introduce new infix operators. For example, the syntax definition system integrated into Standard ML allows the programmer to designate `::` as a right-associative infix operator at precedence level 5 by placing the following directive in the program text:

```
infixr 5 ::
```

This directive causes expressions of the form `e1 :: e2` to desugar to **op**`:: (e1, e2)`, i.e. the variable **op**`::` is applied to the pair `(e1, e2)`. Given that **op**`::` is a list value constructor in SML, this expression constructs a list with head `e1` and tail `e2`.

The fixity directive above also causes patterns of the form `p1 :: p2` to desugar to **op**`:: (p1, p2)`, i.e. to pattern constructor application. Again, because **op**`::` is a list pattern constructor in SML, the desugaring of this pattern matches lists where the head matches `p1` and the tail matches `p2`. (If we had used the identifier `Cons`, rather than **op**`::`, in the definition of the `list` datatype, we would never be able to use the `::` operator in list patterns because SML does not support pattern synonyms.)

[6] In fact, quotation syntax can be expressed using parametric TSMs, which are the topic of Chapter 5.

```
infix 5 ::
infix 6 <*>
infix 4 <|>

functor RXOps(R : RX) =
struct
  structure RU = RXUtil(R)
  val op:: = R.Seq
  val op<*> = RU.repeat
  val op<|> = R.Or
end
```

Figure 2.8: Fixity declarations and related bindings for `RX`

Figure 2.8 shows three fixity declarations related to our regex library together with a functor `RXOps` that binds the corresponding identifiers to the appropriate functions. Assuming that a library packaging system has brought the fixity declarations and the definition of `RXOps` from Figure 2.8 into scope, we can instantiate `RXOps` and then **open** this instantiated module to bring the necessary bindings into scope as follows:

```
structure ROps = RXOps(R)
open ROps
```

We can now draw the previous examples equivalently as follows:

```
val dna = (R.Str "A") <|> (R.Str "T") <|> (R.Str "G") <|>
          (R.Str "C")
val ssn = (RU.digit)<*>3 :: (RU.digit)<*>2 :: (RU.digit)<*>4
fun lookup_rx(name : string) =>
  (Str name) :: (Str ": ") :: ssn
```

This demonstrates two other problems with this approach.

First, it grants only limited control over form – we cannot express the POSIX forms in this way, only *ad hoc* (and in this case, rather poor) approximations thereof.

Second, there can be syntactic conflicts between libraries. Here, both the list library and the regex library have defined a fixity directive for the `::` operator, but each specifies a different associativity. As such, clients cannot use both forms in the same scope. There is no mechanism that allows a client to explicitly qualify an infix operator as referring to the fixity directive from a particular library – fixity directives are not exported from modules or otherwise integrated into the binding structure of SML (libraries are extralinguistic packaging constructs, distinct from modules.)

Formally, each fixity directive induces a dialect of the subset of SML's textual syntax that does not allow the declared identifier to appear in prefix position. When two such dialects are combined, the resulting dialect is not necessarily a dialect of both of the constituent dialects (one fixity declaration overrides the other, according to the order in which the dialects were combined.)

Due to these limitations, VerseML does not inherit this mechanism from SML (the infix operators that are available in VerseML, like `^` for string concatenation, have a fixed precedence, associativity and desugaring.)

2.4.4 Mixfix Syntax Definitions

Fixity directives do not give direct control over desugaring – the desugaring of a binary operator form introduced by a fixity directive is always of function application or pattern constructor application form. "Mixfix" syntax definition systems generalize SML-style fixity directives in that newly defined forms can contain any number of sub-trees (rather than just two) and their desugarings are determined by a programmer-defined rewriting.

The simplest of these systems, e.g. Griffin's system of notational definitions [57], later variations on this system with stronger theoretical properties [120], and the syntax definition system integrated into the Agda programming language [33], support only forms that contain a fixed number of sub-trees, e.g. **if** _ **then** _ **else** _. We cannot define SML-style derived list forms using these systems, because list forms can contain any number of sub-trees.

More advanced notational definition systems support new forms that contain n-ary sequences of sub-trees separated by a given token. For example, Coq's notation system [86] can be used to express list syntax as follows:

```
Notation " [ ] " := nil (format "[ ]") : list_scope.
Notation " [ x ] " := (cons x nil) : list_scope.
Notation " [ x ; y ; .. ; z ] " :=
  (cons x (cons y .. (cons z nil) ..)) : list_scope.
```

Here, the final declaration handles a sequence of $n > 1$ semi-colon separated trees.

Even under these systems, we cannot define POSIX-style regex syntax. The problem is that we can only extend the syntax of the existing sorts of trees, e.g. types, expressions and patterns. We cannot define new sorts of trees, with their own distinct syntax. For example, we cannot define a new sort for regular expressions, where sequences of characters are not recognized as Coq identifiers but rather as regex character sequences.

As with other mechanisms for defining syntax dialects, we cannot reason modularly about syntactic determinism. The Coq manual acknowledges this [86]:

> *Mixing different symbolic notations in [the] same text may cause serious parsing ambiguity.*

To help library clients manage conflicts when they arise, most of these systems include various precedence mechanisms. For example, Agda supports a system of directed acyclic precedence graphs [33] (this is related to earlier work by Aasa where a complete precedence graph was necessary [12].) In Coq, the programmer can associate notation definitions with named "scopes", e.g. `list_scope` in the example above. A scope can be activated or deactivated explicitly using scope directives to control the availability of notation definitions. The innermost scope has the highest precedence. In some situations, Coq is able to use type information to activate a scope implicitly. Mixfix syntax definition systems that use types more directly to disambiguate from several possibilities have also been developed [90, 130]. These only reduce the likelihood of a conflict – they do not eliminate the possibility entirely.

Aasa et al. developed a system whereby each constructor of a datatype definition could have its own syntax [11, 13]. This syntax was delimited from the rest of the language using a fixed quotation-antiquotation system like that described in Sec. 2.4.2.

Parsing was integrated into the type inference mechanism of the language. However, this system is also not expressive enough to handle POSIX regex syntax, again because it forces an immediate, one-to-one correspondence between constructors and syntactic forms. For example, it is not possible to treat arbitrary character sequences as regex character sequences, which are governed by the `Str` constructor. It is also not possible to capture idioms that do not correspond immediately to datatype constructor application (e.g. idioms involving modules.)

2.4.5 Grammar-Based Syntax Definition Systems

Many syntax definition systems are oriented around *formal grammars* [67]. Formal grammars have been studied since at least the time of Pãnini, who developed a grammar for Sanskrit in or around the 4th century BCE [70].

Context-free grammars (CFGs) were first used to define the textual syntax of a major programming language – Algol 60 – by Backus [91]. Since then, countless other syntax definition systems oriented around CFGs have emerged. In these systems a syntax definition consists of a CFG (perhaps from some restricted class of CFGs) equipped with various auxiliary definitions (e.g. a lexer definition in many systems) and logic for computing an output value (e.g. a tree) based on the determined form of the source text.

Perhaps the most established CFG-based syntax definition systems within the ML ecosystem are ML-Lex and ML-Yacc, which are distributed with SML/NJ [121], and Camlp4, which was (until recently) integrated into the OCaml system (in recent releases of the OCaml system, it has been deprecated in favor of a simpler system, `ppx`, that we discuss in the next section) [79]. In these systems, the output is an ML value computed by ML functions that appear associated with productions in the grammar (these functions are referred to as the *semantic actions*.)

The *syntax definition formalism (SDF)* [64] is a syntactic formalism for describing CFGs. SDF is used by a number of syntax definition systems, e.g. the Spoofax "language workbench" [75]. These systems commonly use Stratego, a rule-based rewriting language, as the language that output logic is written in [125]. SugarJ is an extension of Java that allows programmers to define and combine fragments of SDF+Stratego-based syntax definitions directly from within the program text [43]. SugarHaskell is a similar system based on Haskell [44] and Sugar* simplifies the task of defining similar extensions of other languages [41]. SoundExt and SugarFOmega add the requirement that new derived forms must come equipped with derived typing rules [81]. The system must be able to verify that the rewrite rules are sound with respect to these derived typing rules (their verification system defers to the proof search facilities of PLT-Redex [46].) SoundX generalizes this idea to other base languages, and adds the ability to define type-dependent rewritings [82]. We will say more about SoundExt/SugarFOmega and SoundX when we discuss abstract reasoning under syntax dialects below.

Copper implements a CFG-based syntax definition system that uses a context-aware scanner [131]. We will say more about Copper when we discuss modular reasoning about syntactic determinism below.

Some other syntax definition systems are instead oriented around *parsing expression grammars* (PEGs) [50]. PEGs are similar to CFGs, distinguished mainly in that they are deterministic by construction (by allowing only for explicitly prioritized choice between alternative parses.) *Packrat parsers* implement PEGs [51].

2.4.6 Parser Combinator Systems

Parser combinator systems specify a functional interface for defining parsers, together with various functions that generate new parsers from existing parsers and other values (these functions are referred to as the *parser combinators*) [68]. In some cases, the composition of various parser combinators can be taken as definitional (as opposed to the usual view, where a parser is an implementation of a syntax definition.)

For example, Hutton describes a system where parsers are functions of some type in the following parametric type family:

```
type parser('c, 't) = list('c) -> list('t * list('c))
```

Here, a parser is a function that takes a list of (abstract) characters and returns a list of valid parses, each of which consists of an (abstract) output (e.g. a tree) and a list of the characters that were not consumed. An input is ambiguous if this function returns more than one parse. A deterministic parser is one that never returns more than one parse. The non-deterministic choice combinator `alt` has the following signature:

```
val alt : parser('c, 't) -> parser('c, 't) -> parser('c, 't)
```

The `alt` combinator combines the two given parsers by applying them both to the input and appending the lists that they return.

Various alternative designs that better control dynamic cost or that maintain other useful properties have also been described. For example, Hutton and Meijer describe a parser combinator system in monadic style [69]. Okasaki has described an alternative design that uses continuations to control cost [94].

Some systems use a layer of directives placed in the source text to control parser invocation. For example, in Racket's reader macro system, the programmer can direct the initial token reader to shift control to a given parser when a designated directive or token is seen [48, 49]. Honu is another reader based system, which uses a simple syntactic pattern language to initially "enforest" the token stream, i.e. to turn it into a simple tree structure, before passing it to the parser [102].

2.4.7 Examples of Syntax Dialects

Now that we have given an overview of a number of syntax definition systems, let us consider two specific examples of syntax dialects to motivate our subsequent discussion of the problems with the dialect oriented approach.

Example 1: $\mathcal{V}_{\texttt{rx}}$

Using any of the more general syntax definition systems described in the two previous sections, we can define a dialect of VerseML's textual syntax called $\mathcal{V}_{\texttt{rx}}$ that builds in derived regex forms.

```
val ssn = /\d\d\d-\d\d-\d\d\d\d/
fun lookup_rx(name : string) => /@name: %ssn/
```

Figure 2.9: Derived regex expression forms in $\mathcal{V}_{\texttt{rx}}$

```
fun is_dna_rx(r : rx) : boolean =>
  match r with
  | /A/ => True
  | /T/ => True
  | /G/ => True
  | /C/ => True
  | /%(r1)%(r2)/ => (is_dna_rx r1) andalso (is_dna_rx r2)
  | /%(r1)|%(r2)/ => (is_dna_rx r1) andalso (is_dna_rx r2)
  | /%(r)*/ => is_dna_rx r'
  | _ => False
  end
```

Figure 2.10: Derived regex pattern forms in $\mathcal{V}_{\texttt{rx}}$

In particular, $\mathcal{V}_{\texttt{rx}}$ extends the syntax of expressions with *derived regex literals*, which are delimited by forward slashes, e.g. `/A|T|G|C/`. The desugaring of this form is equivalent to the following if we assume that `Or` and `Str` stand for the corresponding constructors of the recursive labeled sum type `rx` that was defined in Figure 2.2:

```
Or(Str "A", Or (Str "T", Or (Str "G", Str "C")))
```

Of course, it is unreasonable to assume that `Or` and `Str` are bound appropriately at every use site. In order to maintain *context independence*, the desugaring instead applies the explicit **`fold`** and **`inj`** operators as discussed in Sec. 2.3.1.[7]

$\mathcal{V}_{\texttt{rx}}$ also supports regex literals that contain subexpressions. These capture the idioms that arise when constructing regex values compositionally. For example, the definition of `lookup_rx` in Figure 2.9 is equivalent to the definition of `lookup_rx` that was given in Figure 2.7. The prefix `@` followed by the identifier `name` causes the expression `name` to appear in the desugaring as if wrapped in the `Str` constructor, and the prefix `%` followed by the identifier `ssn` causes `ssn` to appear in the desugaring directly. We refer to the expressions that appear inside literal forms as *spliced expressions*.

To splice in an expression that is not of variable form, e.g. a function application, we must delimit it with parentheses: `/@(capitalize name)/`.

Finally, $\mathcal{V}_{\texttt{rx}}$ extends the syntax of patterns with analagous *derived regex pattern literals*. For example, the definition of `is_dna_rx` in Figure 2.10 is equivalent to the definition of `is_dna_rx` that was given in Figure 2.3. Notice that the variables bound by the patterns in Figure 2.10 appear inside *spliced patterns*.

[7] In SML, where datatypes are abstract and explicit fold and injection operators are not exposed, it is more difficult to maintain context independence. We would need to provide a module containing the constructors as a "syntactic argument" to each form – we describe this technique as it relates to our modular encoding of regexes in Example 2 below.

```
fun is_dna_rx'(r : R.t) : boolean =>
  match R.unfold_norm r with
  | /A/ => True
  | /T/ => True
  | /G/ => True
  | /C/ => True
  | /%(r1)%(r2)/ => (is_dna_rx' r1) andalso (is_dna_rx' r2)
  | /%(r1)|%(r2)/ => (is_dna_rx' r1) andalso (is_dna_rx' r2)
  | /%(r)*/ => is_dna_rx r'
  | _ => False
  end
```

Figure 2.11: Derived regex unfolding pattern forms in $\mathcal{V}_{\text{RX}}$

Example 2: $\mathcal{V}_{\text{RX}}$

In Sec. 2.3.2, we also considered a more sophisticated formulation of our regex library organized around the signature `RX` defined in Figure 2.4. Let us define another dialect of VerseML's textual syntax called $\mathcal{V}_{\text{RX}}$ that defines derived forms whose desugarings involve modules that implement `RX`. For this to work in a context-independent manner, these forms must take the particular module that is to appear in the desugaring as a spliced subterm. For example, in the following program fragment, the module `R` is "passed into" each derived form for use in its desugaring:

```
val ssn = R./\d\d\d-\d\d\d\d-\d\d\d/
fun lookup_rx'(name : string) => R./@name: %ssn/
```

The desugaring of the body of `lookup_rx'` is:

```
R.Seq(R.Str(name), R.Seq(R.Str ": ", ssn))
```

This desugaring logic is context-independent because the constructors are explicitly qualified (i.e. `Seq` and `Str` are *component labels* here, not variables.) The only variables that appear in the desugaring are `R`, `name` and `ssn`. All of these were specified by the client at the use site, so they are subject to renaming.

Recall that `RX` specifies a function `unfold_norm : t -> u(t)` for computing the normal unfolding of the given regex. $\mathcal{V}_{\text{RX}}$ defines derived forms for patterns matching values of types in the type family `u('a)`. These are used in the definition of `is_dna_rx'` given in Figure 2.11.

2.4.8 Problems with Syntax Dialects

Conservatively Combining Syntax Dialects

Notice that the derived regex pattern forms that appear in Figure 2.11 are identical to those that appear in Figure 2.10. Their desugarings are, however, different. In particular, the patterns in Figure 2.11 match values of type `u('a)`, whereas the patterns in Figure 2.10 match values of type `rx`.

```
fun is_dna_rx'(r : R.t) : boolean =>
  match R.unfold_norm r with
   | $cmu_edu_comar_rx $u/A/ => True
   | $cmu_edu_comar_rx $u/T/ => True
   | $cmu_edu_comar_rx $u/G/ => True
   | $cmu_edu_comar_rx $u/C/ => True
   (* and so on *)
   | _ => False
  end
```

Figure 2.12: Using URI-based grammar names together with marking tokens to avoid syntactic conflicts

It would be useful to have derived forms for values of type `rx` available even when we are working with the modular encoding of regexes, because we have defined a function `view : R.t -> rx` in `RXUtil`. This brings us to the first of the two main problems with the dialect-oriented approach, already described in Chapter 1: there is no good way to conservatively combine $\mathcal{V}_{\text{rx}}$ and $\mathcal{V}_{\text{RX}}$. In particular, any such "combined dialect" will either fail to conserve determinism (because the forms overlap), or the combined dialect will not be a dialect of both of the constituent dialects, i.e. some of the forms from one dialect will "shadow" the overlapping forms from the other dialect (depending on the order in which they were combined [50].)

In response to this problem, Schwerdfeger and Van Wyk have developed a modular analysis that accepts only deterministic extensions of a base LALR(1) grammar where all new forms must start with a "marking" terminal symbol and obey certain other constraints related to the follow sets of the base grammar's non-terminals [106]. By relying on a context-aware scanner (a feature of Copper [131]) to transfer control when the marking terminals are seen, extensions of a base grammar that pass this analysis and specify disjoint sets of marking terminals can be combined without introducing conflict.

For the two dialects just considered, these conditions are not satisfied. If we modify the grammar of $\mathcal{V}_{\text{RX}}$ so that, for example, the regex literal forms are marked with `$r` and the regex unfolding forms are marked with `$u`, the analysis will accept both grammars, and the combine-time disjointness check will pass, solving our immediate problem at only a small cost. However, a conflict could still arise later when a client combines these extensions with another extension that also uses the marking terminals `$r`, `$u` or `/`.

The solution proposed by Schwerdfeger and van Wyk [106] is 1) to allow for the grammar's name to be used as an additional syntactic prefix when a conflict arises, and 2) to adopt a naming convention for grammars based on the Internet domain name system (or some similar coordinating system) that makes conflicts unlikely. For example, Figure 2.12 shows how a client would need to draw `is_dna_rx'` if a conflict arose. Clearly, this drawing has higher syntactic cost than the drawing in Figure 2.11. Moreover, there is no simple way for clients to selectively control this cost by defining scoped abbreviations for marking tokens or grammar names (as one does for types, modules or values that are exported from deeply nested modules) because this mechanism is purely syntactic, i.e. agnostic to the binding structure of the language.

Another approach aimed at making conflicts less likely, though not impossible, is to use types to choose from amongst several possible parses. Some approaches require generating the full *parse forest* before typechecking proceeds, e.g. the *MetaBorg* system [21]. This approach is inefficient, particularly when a large number of grammars have been composed. The method of *type-oriented island parsing* integrates parsing and typechecking so that disambiguation occurs as early as possible [112].

A more radical approach would be to insist that programmers use a *language composition editor* like Eco [36]. Language composition editors allow programmers to explicitly switch from one syntax to another with an editor command. This is an instance of the more general concept of *structure editing* (also called *structured editing*, *projectional editing* or *syntax-directed editing*.) This concept, pioneered by the Cornell Program Synthesizer [122], has various costs and benefits, summarized in [126]. In this work, our interest is in text-based syntax, but we consider structure editors as future work in Sec. 8.2.

Abstract Reasoning About Derived Forms

In addition to the difficulties of conservatively combining syntax dialects, there are a number of other difficulties related to the fact that there is often no useful notion of syntactic abstraction that a programmer can rely on to reason about an unfamiliar derived form. The programmer may need to examine the desugaring, the desugaring logic or even the definitions of all of the constituent dialects, to definitively answer the questions given in Sec. 1.2.1. These questions were stated relative to a particular example involving the query processing language K. Here, we generalize from that example to develop an informal classification of the properties that programmers might have difficulty reasoning about in analagous situations. In each case, we will discuss exceptional systems where these difficulties are ameliorated or avoided entirely.

Responsibility It is not always straightforward to determine which constituent dialect is responsible for any particular derived form.

The system implemented by Copper [106] is an exception, in that the marking terminal (and the grammar name, if necessary) allows clients to search across the constituent dialect definitions for the corresponding declaration without needing to understand any of them deeply.

Segmentation It is not always possible to segment a derived form such that each segment consists either of a spliced base language term (which we have drawn in black in the examples in this document) or a sequence of characters that are parsed otherwise (which we have drawn in color.) Even when a segmentation exists, determining it is not always straightforward.

For example, consider a production in a grammar that looks like this:

```
start <- "%(" verseml_exp ")"
```

The name of the non-terminal `verseml_exp` suggests that it will match any VerseML expression, but it is not certain that this is the case. Moreover, even if we know that this non-terminal matches VerseML expressions, it is not certain that the output logic will insert that expression as-is into the desugaring – it may instead only examine its form, or transform it in some way (in which case highlighting it as a spliced expression might be misleading.)

Systems that support the generation of editor plug-ins, such as Spoofax [75] and Sugarclipse for SugarJ [42], can generate syntax coloring logic from an annotated grammar definition, which often give programmers some indication of where a spliced term occurs. However, there is no definitive information about segmentation in how the editor displays the derived form. (Moreover, these editor plug-ins can themselves conflict, even if the syntax itself is deterministic.)

Capture The desugaring of a derived form might place spliced terms under binders. These binders are not visible in the program text, but can shadow those that are. As a result, the spliced terms will inadvertently capture these expansion-internal bindings. This significantly obscures the binding structure of the program.

For derived forms that desugar to module-level definitions (e.g. to one or more **`val`** definitions), a desugaring might also introduce exported module components that are similarly invisible in the text. This can cause non-local capture when a client **`open`**s that module into scope.

In most cases, capture is inadvertent. For example, a desugaring might bind an intermediate value to some temporary variable, `tmp`. This can cause problems at use sites where `tmp` is bound. It is easy to miss this problem in testing (particularly if the types of both bindings are compatible.)

In some syntax dialects, capture is by design. For example, in (Sugar)Haskell, **`do`** notation for monadic values operates as a new binding construct [44]. For programmers who are familiar with **`do`** notation, this can be useful. But when a programmer encounters an unfamiliar form, this forces them to determine whether it similarly is designed as a new binding construct. A simple grammar provides no information about capture.

In most systems, it is possible for dialect providers to generate identifiers that are guaranteed to be fresh at the use site. If dialect providers are disciplined about using this mechanism, they can prevent capture. However, this is awkward and most systems provide no guarantee that the dialect provider maintained this freshness discipline [45].

To enforce a prohibition on capture, the system must be integrated into or otherwise made aware of the binding structure of the language. For example, some of the language-integrated mixfix systems discussed above, e.g. Coq's notation system [86], enforce a prohibition on capture by alpha-renaming desugarings as necessary. Erdweg et al. have developed a formalism for directly describing the "binding structure" of program text, as well as contextual transformations that use these descriptions to rename the identifiers that appear in a desugaring to avoid capture [45, 105].

Context Dependence If the desugaring of a derived form assumes that certain identifiers are bound at the application site (e.g. to particular values, or to values of some particular type), we refer to the desugaring as being *context dependent*.

Context dependent desugarings take control over naming away from clients. Moreover, it is difficult to determine the assumptions that a desugaring is making. As such, it becomes difficult to reason about whether renaming an identifier or moving a binding is a meaning-preserving transformation.

In our examples above, we maintained context independence as a "courtesy" by explicitly applying the **`fold`** and **`inj`** operators, or by taking the module for use in the desugaring as a "syntactic argument".

To enforce context independence, the system must be aware of binding structure and have some way to distinguish those subterms of a desugaring that originate in the text at the use site (which should have access to bindings at the use site) from those that do not (which should only have access to bindings internal to the desugaring.) For example, language-integrated mixfix systems, e.g. Coq's notation system, use a simple rewriting system to compute desugarings, so they satisfy these requirements and can enforce context independence. Coq gives desugarings access only to the bindings visible where the notation was defined.

More flexible systems where desugarings are computed functionally, or language-external systems that have no understanding of binding structure, do not enforce context independence.

Typing Finally, it is not always clear what type an expression drawn in derived form has, or what type of value that a pattern drawn in derived form matches. Similarly, it is not always straightforward to determine what type a spliced expression has, or what type of value that a spliced pattern matches.

SoundExt/SugarFomega [82] and SoundX [105] allow dialect providers to define derived typing rules alongside derived forms and desugaring rules. These systems automatically verify that the desugaring rules are sound with respect to these derived typing rules. This ensures that type errors are never reported in terms of the desugaring (which is the stated goal of their work.) However, this helps only to a limited extent in answering the questions just given. In particular, the programmer must first assign Responsibility (which is difficult for the reasons just given.) Next, the programmer must identify the spliced terms (which is difficult because these systems to not make it easy to reason about Segmentation, as just described.) Then, the programmer must construct a derivation using the relevant derived typing rules. Finally, the programmer must traverse the derivation to find out where the spliced terms appear within it to answer questions about their type. Even for relatively simple base languages, like System $\mathbf{F}_\omega$, understanding a typing derivation requires significantly more effort and expertise than programmers usually need.[8] For languages like ML, the judgement forms are substantially more complex (no one has yet attempted to apply the SoundX methodology to a language as large as ML.)

Systems like MetaBorg that require that the type of a derived form be known from

[8]At CMU, we teach ML to all first-year students (in 15-150 – Functional Programming.) However, understanding a judgmental specification of a language like System $\mathbf{F}_\omega$ involves skills that are taught only to some third and fourth year students (in 15-312 – Principles of Programming Languages.)

context so that disambiguation can occur (see above) also address the problem of determining the type of a derived expression or pattern form as a whole. However, it is not always clear what the types of the spliced terms within these derived forms should be.

2.4.9 Non-Local Term Rewriting Systems

Another approach is to leave the textual syntax of the language fixed, but repurpose it for novel ends using a *term rewriting system*. Term rewriting systems transform syntactically well-formed terms into other syntactically well-formed terms (unlike syntax definition systems, which operate on the program text.)

Non-local term rewriting systems typically operate over an entire compilation unit (e.g. a file). For example, one could define a preprocessor that rewrites every string literal that is followed by the comment `(*rx*)` to the corresponding expression (or pattern) of type `rx`. For example, the following expression would be rewritten to a regex expression, with `dna` treated as a spliced subexpression as described in the previous section:

```
"GC%(dna)GC"(*rx*)
```

OCaml 4.02 introduced *preprocessor extension (ppx) points* into its textual syntax [79]. Extension points serve as markers for the benefit of a non-local term rewriting system. They are less *ad hoc* than comments, in that each extension point is associated with a single term in a well-defined way, and the compiler gives an error if any extension points remain after preprocessing is complete. For example, in the following program fragment,

```
let%lwt (x, y) = f in x + y
```

the `%lwt` annotation on the let expression is recognized by a preprocessor distributed with `Lwt`, a lightweight threading library. This preprocessor rewrites this fragment to:

```
Lwt.bind f (fun (x, y) -> x + y)
```

The OCaml system is distributed with a library called `ppx_tools` that simplifies the task of writing preprocessors that operate on terms annotated with extension points.

There are a number of other systems that support non-local term rewriting. For example, the `GHC` compiler for Haskell [73] and the `xoc` compiler for C [29] both support user-defined non-local rewritings.

These systems present several difficulties with abstract reasoning, many of which are directly analagous to those that syntax definition systems present:

1. **Conflict:** Different preprocessors may recognize the same markers or code patterns.
2. **Responsibility:** It is not always clear which preprocessor handles each rewritten form.
3. **Localization:** A non-local term rewriting system might insert code anywhere in the program, complicating reasoning efforts.
4. **Segmentation:** It is not always clear where spliced terms appear inside rewritten string literal forms.
5. **Capture:** The rewriting might place terms under binders that shadow bindings visible in the program text.

6. **Context Dependence:** The rewriting might assume that certain identifiers are bound at particular locations, making it difficult to reason about refactoring.
7. **Typing:** It is not always clear what type the rewriting of a marked form will have (if indeed the rewriting happens to be local.) Similarly, the type that terms that appear within the rewritten form should have is often unclear.

2.4.10 Term-Rewriting Macro Systems

Macro systems are language-integrated local term rewriting systems, i.e. they allow programmers to designate functions that implement rewritings as macros. Clients apply macros directly to terms (e.g. expressions, patterns and other sorts of terms.). The rewritten term is known as the *expansion* of the macro application.

Macro systems do not suffer from problems related to reasoning about **Conflict**, **Responsibility** and **Localization** described above because macros are applied explicitly and operate locally.

Naïve macro systems, like the earliest variants of the LISP macro system [63], early compile-time quotation expanders in ML [85], Template Haskell macros [111] and GHC quasiquotes [84], do not escape from the remaining problems described above, because they can generate arbitrary code for insertion at the macro application site. For example, it is possible in early LISP dialects and in these other less disciplined modern macro systems to define a macro `rx!` that can be applied to rewrite a string form containing a spliced subexpression to a regex:

```
(rx! "GC%(dna)GC")
```

The problem with these systems is that without examining the macro's implementation or the generated expansion, there is no way to reason about **Segmentation**, **Capture**, **Context Dependence** or **Typing**.[9]

The problem of **Capture** was addressed by the design of Scheme's *hygienic macro system* [15, 28, 40, 65, 66, 76], which automatically alpha-renames identifiers bound in the expansion so that they do not shadow those that appear at the macro application site.

The problem of **Context Dependence** is typically confronted by allowing macro expansions to explicitly refer only to those bindings in scope at the macro definition site. These references are preserved even if the identifiers involved have been shadowed at the macro application site [15, 28, 40]. Any references to application site bindings must originate in one of the macro's arguments. There are two problems with this approach:

1. It does not make explicit which of the definition site bindings the expansions generated by a macro might refer to, so reasoning abstractly about the renaming of definition site bindings remains problematic.
2. Preventing access to the application site bindings makes defining a macro like `rx!` impossible, because spliced subexpressions (like `dna` above) do not appear as subexpressions of an argument to `rx!` – they are parsed out of a string literal

[9]It is not enough that the generated expansions be typechecked – it must be possible for the user to reason about *what the type of the expansion is*.

programmatically. From the perspective of the macro system, such spliced subexpressions are indistinguishable from inappropriate references to bindings tracked by the application site context.

The only choice, then, is to repurpose other forms that do contain subexpressions. For example, the macro might repurpose infix operators that usually have a different meaning, e.g. `^`:

```
(rx! ("GC" ^ dna ^ "GC"))
```

This is rather confusing, in that it appears that string concatenation is occuring when that is not the case – `rx!` is simply repurposing the infix `^` form.

The problem of reasoning about **Typing** is relatively understudied, because most research on macro systems has been done in languages in the LISP tradition that do not define a rich static semantics.

Herman and Wand's calculus of macros [65, 66] does use a type system to reason about the binding structure of the expansion that a macro generates, but the expansions themselves are not written in a language with rich type structure.

Some macro systems for languages with non-trivial type structure, like Template Haskell [111], do not support reasoning about types in that the guarantee is only that the expansion is well-typed – clients cannot reason about *what that type is*.

Other macro systems, like MacroML [53, 109], support reasoning about typing, but these systems are *staging macro systems*, rather than *term-rewriting macro systems*, meaning that the macro does not have access to the syntax tree of the arguments at all. Staging macros cannot be used for syntactic control – macro application syntactically coincides with function application. These macro systems are instead motivated primarily by concerns about performance.

The Scala macro system is a notable example of a term-rewriting macro system that does allow reasoning about typing [23]. In particular, Scala's "black box" macros include type annotations on the arguments. We are not aware of a typed macro system that has been integrated into a language with an ML-style module system. The main problem with Scala's macro system, then, is that it does not give us enough syntactic control – we must repurpose Scala's existing syntactic forms, as discussed in point 2 above.

Chapter 3

Simple Expression TSMs (seTSMs)

In the remainder of this work, we will develop a system of *typed syntax macros (TSMs)*. Briefly, TSMs offer substantially greater syntactic flexibility as compared to typed term rewriting macros *a la* Scala, while 1) guaranteeing that a segmentation can always be produced; 2) enforcing a prohibition on capture; 3) enforcing a strong form of context independence and 4) maintaining the ability to reason abstractly about types. We will establish these reasoning principles formally, ultimately in a system with an ML-style module system in Chapter 5. We will begin, however, in this chapter with a simpler calculus of expressions and types. The TSMs available in this calculus are called *simple expression TSMs* (seTSMs).

3.1 Simple Expression TSMs By Example

We begin in this section with a "tutorial-style" introduction to seTSMs in VerseML. Sec. 3.2 then formally defines a reduced dialect of VerseML called $\mathsf{miniVerse}_{\mathbf{SE}}$. This will serve as a "conceptually minimal" core calculus of TSMs, in the style of the simply typed lambda calculus.

3.1.1 TSM Application

The following VerseML expression, drawn textually, is of *TSM application* form. Here, a TSM named `$rx` is applied to the *generalized literal form* `/A|T|G|C/`:

```
$rx /A|T|G|C/
```

Generalized literal forms are left unparsed according to the context-free syntax of VerseML. Several other outer delimiters are also available, as summarized in Figure 3.1. The client is free to choose any of these for use with any TSM, as long as the *literal body* (shown in green above) satisfies the requirements stated in Figure 3.1. For example, we could have equivalently written the example above as `$rx `A|T|G|C``. (In fact, this would have been convenient if we had wanted to express a regex containing forward slashes but not backticks.)

```
'body cannot contain an apostrophe'
`body cannot contain a backtick`
[body cannot contain unmatched square brackets]
{|body cannot contain unmatched barred curly braces|}
/body cannot contain a forward slash/
\body cannot contain a backslash\
```

Figure 3.1: Generalized literal forms available for use in VerseML's textual syntax. The characters in green indicate the literal bodies and describe how the literal body is constrained by the form shown on that line. The Wyvern language defines additional forms, including whitespace-delimited forms [96] and multipart forms [97], but for simplicity we leave these out of VerseML.

It is only during the subsequent *typed expansion* phase that the applied TSM parses the body of the literal form to generate a *proto-expansion*. The language then *validates* this proto-expansion according to criteria that we will describe in Sec. 3.1.5. If proto-expansion validation succeeds, the language generates the *final expansion* (or more concisely, simply the *expansion*) of the TSM application. The behavior of the program is determined by its expansion.

For example, the expansion of the TSM application above is equivalent to the following expression when the regex value constructors `Or` and `Str` are in scope:

```
Or(Str "A", Or(Str "T", Or(Str "G", Str "C")))
```

To avoid the assumption that the variables `Or` and `Str` are in scope at the TSM application site, the expansion actually uses the explicit **fold** and **inj** operators, as described in Sec. 2.3.1. In fact, the proto-expansion validation process enforces this notion of context independence – we will return to proto-expansion validation below. (We will show how TSM parameters can reduce the awkwardness of this requirement in Chapter 5.)

3.1.2 TSM Definitions

The definition of `$rx` takes the following form:

```
syntax $rx at rx by
  static fn(b : body) -> parse_result(proto_expr) =>
    (* regex literal parser here *)
end
```

Every seTSM definition consists of a TSM name, here `$rx`, a *type annotation*, here **at** `rx`, and a *parse function* between **by** and **end**. TSM definitions follow standard scoping rules – unless an **in** clause is provided, the definition is in scope until the end of the enclosing declaration (e.g. the enclosing function or module.) We will consider how TSM definitions are packaged into libraries in Chapter 6.

All TSM names must begin with the dollar symbol (`$`), which distinguishes them from variables. This is inspired by the Rust macro system, which uses post-fix exclamation points (`!`) to distinguish macro identifiers [5].

```
type body = string

type segment = {startIdx : int, endIdx : int} (* inclusive *)
type parse_result('a) = ParseError of {
                          msg : string, loc : segment
                        }
                      + Success of 'a
```

Figure 3.2: Definitions of `body`, `segment` and `parse_result`. These type definitions are given in the VerseML *prelude*, which is a small collection of definitions available ambiently.

The parse function is a *static function* delegated responsibility over parsing the literal bodies of the literal forms to which the TSM 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 do not themselves make use of TSMs (we will eliminate these impractical limitations in Chapter 6.)

Every seTSM 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 3.2.[1] The return type is a labeled sum type, defined by applying the type function `parse_result` defined in Figure 3.2, that distinguishes between parse errors and successful parses.[2] 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:

inj[ParseError]({msg=e_{msg}, loc=e_{loc}})

where e_{msg} is an error message and e_{loc} is a value of type `segment`, defined in Figure 3.2, that designates a segment of the literal body as the location of the error [124]. This information is for use by VerseML compilers when reporting the error to the programmer.

Successful Parses If parsing succeeds, the parse function returns

inj[Success](e_{proto})

where e_{proto} is called the *encoding of the proto-expansion*.

For expression TSMs, proto-expansions are *proto-expressions*, which are encoded as VerseML values of the type `proto_expr` defined in Figure 3.3. Most of the variants defined by `proto_expr` are individually uninteresting – they encode VerseML's various expression

[1] In languages where the surface syntax is not textual, `body` would have a different definition, but we leave explicit consideration of such languages as future work (see Sec. 8.2.)

[2] `parse_result` is defined as a type function because in Chapter 4, we will introduce pattern TSMs, which generate patterns rather than expressions.

```
type proto_typ = rec(proto_typ =>
                   TyVar of var_t
                 + Arrow of proto_typ * proto_typ
                 + (* ... *)
                 + SplicedT of segment)

type proto_expr = rec(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)
```

Figure 3.3: Abbreviated definitions `proto_typ` and `proto_expr` in the VerseML prelude. We assume some suitable type `var_t` exists, not shown.

forms (just as in a compiler, c.f. SML/NJ's Visible Compiler library [114].) Expressions can mention types, so we also need to define a type `proto_typ` in Figure 3.3. As we enrich our language in later chapters, we will need to define more encodings like these, for other sorts of trees. The only non-standard classes are `SplicedT` and `SplicedE` – these are *references to spliced unexpanded types and expressions*, which we will return to when we consider splicing in Sec. 3.1.3 below.

The definitions of `proto_typ` and `proto_expr` are recursive labeled sum types to simplify our exposition, but we could have chosen alternative encodings, e.g. based on abstract binding trees [61], with only minor modifications to our semantics. Indeed, when we formally define seTSMs in Sec. 3.2, we abstract over the particular encoding.

3.1.3 Splicing

As described thusfar, TSMs operate just like term-rewriting macros over string literals. TSMs therefore do not cause difficulties related to reasoning about **Conflict**, **Responsibility** or **Localization**, for exactly the reasons discussed in Sec. 2.4.10. TSMs differ from term-rewriting macros in that they support *splicing out arbitrary types and expressions* (including those that may themselves involve TSM applications) from within literal bodies in a reasonable manner. For example, the program fragment from Figure 2.9 can be expressed using the `$rx` TSM as follows:

```
val ssn = $rx /\d\d\d-\d\d-\d\d\d\d/
fun lookup_rx(name: string) => $rx /@name: %ssn/
```

The expressions `name` and `ssn` on the second line appear spliced within the literal body, so we call them *spliced expressions*.

When `$rx`'s parse function determines that a subsequence of the literal body should be taken as a spliced expression (here, by recognizing the characters `@` or `%` followed by a variable or parenthesized expression), it does not directly insert the syntax tree of that expression into the encoding of the expansion. Instead, the TSM must refer to the spliced expression *by its relative location* within the literal body using the `SplicedE` variant of `proto_expr`. In particular, the `SplicedE` variant requires a value of type `segment`, which indicates the zero-indexed location of the spliced expression relative to the start of the literal body provided to the parse function. The `SplicedE` variant also requires a value of type `proto_typ`, which indicates the type that the spliced expression is expected to have. For example, the proto-expansion generated by `$rx` for the literal body on the second line above, if written in a textual syntax for proto-expressions where references to spliced expressions are **spliced**`<startIdx; endIndex; ty>`, is:

```
Seq(Str(spliced<1; 4; string>),
    Seq(Str ": ", spliced<8; 10; rx>))
```

Here, **spliced**`<1; 4; string>` refers to the spliced string expression `name` by location and **spliced**`<8; 10; rx>` refers to the spliced regex expression `ssn` by location. (For clarity of exposition, we again use the regex value constructors to abbreviate applications of the **`fold`** and **`inj`** operators and use the type abbreviation `rx`. In fact, given only the mechanisms introduced in this chapter, these abbreviations would need to be explicitly included in each proto-expansion.)

Proto-types can make reference to spliced types by using the `SplicedT` variant of `proto_typ` analagously.

Requiring that the TSM refer to spliced terms indirectly in this manner prevents it from "forging" a spliced expression (i.e. claiming that an expression is a spliced expression when it does not appear in the literal body.) This will be formally critical to being able to reason abstractly about segmentation, capture and context-independence, as we will detail below.

3.1.4 Splice Summaries and Segmentations

The *splice summary* of a proto-expression is the finite set of references to spliced terms within the proto-expression. For example, the summary of the proto-expression above is the finite set containing only **spliced**`<1; 4; string>` and **spliced**`<8; 10; rx>`.

The *segmentation* of a proto-pression is the finite set of locations extracted from the splice summary. For example, the segmentation of the proto-expansion above is:

$$\{(1,4),(8,10)\}$$

The semantics checks that all of the locations in the segmentation are 1) in bounds relative to the literal body; and 2) non-overlapping. This resolves the problem of **Segmentation** described in Secs. 2.4.9-2.4.10, 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:

```
val ssn = $rx /\d\d\d-\d\d-\d\d\d\d/
fun lookup_rx(name: string) => $rx /@name: %ssn/
```

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].)

3.1.5 Proto-Expansion Validation

Three potential problems described in Secs. 2.4.9-2.4.10 remain: those related to reasoning abstractly about **Capture**, **Context Dependence** and **Typing**. Addressing these problems is the purpose of the *proto-expansion validation* process.

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, suppose that `$rx` generated a proto-expansion of the following form (drawn as above):

```
let tmp = (* ... expansion-internal tmp ... *) in
Seq(tmp, spliced<1; 3; rx>)
```

Naïvely, the binding of the variable `tmp` here could shadow bindings of `tmp` that appear at the application site within the indicated spliced expression. For example, consider the following application site:

```
let tmp = (* ... application site tmp ... *) in
$rx /%tmp/
```

Here, the application site binding of `tmp` would be shadowed by the "invisible" binding of `tmp` in the expansion of the TSM application.

To address this problem, proto-expansion validation enforces a prohibition on capture. This prohibition on capture can be silently enforced by implicitly alpha-varying the bindings in the proto-expansion as needed, as in hygienic term-rewriting macro systems (cf. Sec. 2.4.10.) For example, the expansion of the example above might take the following form:

```
let tmp = (* ... application site tmp ... *) in
let tmp' = (* ... expansion-internal tmp ... *) in
Seq(tmp', 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 TSM 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 user-facing mechanism that generates "fresh variables".

TSM 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 defining alternative binding forms. 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 example when we consider other possible points in this design space in Sec. 8.2.10.

Context Dependence

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

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

```
syntax $bad1 at rx by
  static fn(_) => Success (Var "x")
end
```

The proto-expansion that this TSM generates (for every literal body) refers to a variable `x` that is not bound within the expansion. 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 `$bad1` 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. The client's ability to hold the expansion abstract is retained. We saw examples of spliced terms that referred to variables bound at the application site – `name` and `ssn` – in Sec. 3.1.3. Because proto-expansions refer to spliced terms indirectly, and forging is impossible, enforcing context independence is straightforward – we need only that the proto-expansion itself be closed, without considering the spliced terms.

This prohibition on context dependence explains why the expansion generated by the TSM application in Sec. 3.1.1 cannot make use of the regex value constructors, e.g. `Str` and `Or`, directly. (In Chapter 5, we will relax this restriction to allow proto-expansions to access explicit parameters.)

Collectively, we refer to the prohibition on capture and the prohibition on context dependence as *hygiene properties*, by conceptual analogy to corresponding properties in term-rewriting macro systems (see Sec. 2.4.10.) The novelty here comes from the fact that spliced terms are being extracted from an initially unparsed sequence of characters.

Typing

Finally, proto-expansion validation maintains a reasonable *typing discipline* by:

1. checking each spliced expression against the type indicated in the summary; and

2. checking to ensure that the generated expansion is of the type specified in the TSM's type annotation. For example, the type annotation on `$rx` is **`at`** `rx`, so proto-expansion validation ensures that the final expansion is of type `rx`.

This addresses the problem of reasoning abstractly about **Typing** described in Secs. 2.4.9-2.4.10, i.e.:

1. determining the type that a spliced expression must have requires only the information in the summary of the proto-expansion (rather than complete knowledge of the proto-expansion); and
2. determining the type of an expansion requires examining only the type annotation on the TSM definition (much as determining the type of a function application requires examining only the type of the function.)

3.1.6 Final Expansion

The result of proto-expansion validation is the *final expansion*, which is simply the proto-expansion with references to spliced terms replaced with their own final expansions. For example, the final expansion of the body of `lookup_rx` is equivalent to the following, assuming that the regex value constructors were defined (not shown):

```
Seq(Str(name), Seq(Str ": ", ssn))
```

3.1.7 Comparison to the Dialect-Oriented Approach

Let us compare the VerseML TSM `$rx` to $\mathcal{V}_{\texttt{rx}}$, the hypothetical syntactic dialect of VerseML with support for derived forms for values of type `rx` described in Sec. 2.4.7.

Both $\mathcal{V}_{\texttt{rx}}$ and `$rx` give programmers the ability to use the same standard POSIX syntax for constructing regexes, extended with the same syntax for splicing in strings and other regexes. Using `$rx`, however, we incur the additional syntactic cost of explicitly applying the `$rx` TSM each time we wish to use regex syntax. This cost does not grow with the size of the regex, so it would only be significant in programs that involve a large number of small regexes (which do, of course, exist.) In Chapter 7 we will consider a design where even this syntactic cost can be eliminated in positions where the type is known to be `rx`.

The benefit of the TSM-based approach is that we can easily define other TSMs to use alongside the `$rx` TSM without needing to consider the possibility of syntactic conflict. Furthermore, programmers can rely on the binding discipline and the typing discipline enforced by proto-expansion validation to reason about programs, including those that contain unfamiliar forms. Put pithily, VerseML helps programmers avoid "conflict and confusion".

3.2 miniVerse$_{\text{SE}}$

To make the intuitions developed in the previous section precise, we will now introduce a reduced dialect of VerseML called miniVerse$_{\text{SE}}$ that supports seTSMs. The full definition

Sort			Operational Form	Description
Typ	τ	$::=$	t	variable
			$\texttt{parr}(\tau;\tau)$	partial function
			$\texttt{all}(t.\tau)$	polymorphic
			$\texttt{rec}(t.\tau)$	recursive
			$\texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})$	labeled product
			$\texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i\in L})$	labeled sum
Exp	e	$::=$	x	variable
			$\texttt{lam}\{\tau\}(x.e)$	abstraction
			$\texttt{ap}(e;e)$	application
			$\texttt{tlam}(t.e)$	type abstraction
			$\texttt{tap}\{\tau\}(e)$	type application
			$\texttt{fold}(e)$	fold
			$\texttt{unfold}(e)$	unfold
			$\texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L})$	labeled tuple
			$\texttt{prj}[\ell](e)$	projection
			$\texttt{inj}[\ell](e)$	injection
			$\texttt{case}[L](e;\{i \hookrightarrow x_i.e_i\}_{i\in L})$	case analysis

Figure 3.4: Syntax of the miniVerse$_{\mathbf{SE}}$ expanded language (XL)

of miniVerse$_{\mathbf{SE}}$ is given in Appendix B for reference. In the exposition below, we will reproduce only particularly noteworthy rules and proof cases. Rule and theorem numbers below refer to corresponding rules and theorems in the appendix.

3.2.1 Overview

miniVerse$_{\mathbf{SE}}$ consists of a language of *unexpanded expressions* (the *unexpanded language*, or *UL*) defined by typed expansion to a language of *expanded expressions* (the *expanded language*, or *XL*.) We will begin with a brief overview of the standard XL before turning our attention to the UL in the remainder of this chapter.

3.2.2 Syntax of the Expanded Language

The syntax chart in Figure 3.4 defines the syntax of *types*, τ, and *(expanded) expressions*, e. Metavariables x range over expression variables, t over type variables, ℓ over labels and L over finite sets of labels. Types and expanded expressions are ABTs identified up to α-equivalence in the usual manner (our typographic conventions are adapted from *PFPL*, and summarized in Appendix A.1.) To emphasize that programmers never draw expanded terms directly, and to clearly distinguish expanded terms from unexpanded terms, we do not define a stylized or textual syntax for expanded terms.

The XL forms a standard pure functional language with support for partial functions, quantification over types, recursive types, labeled product types and labeled sum types.

The reader is directed to *PFPL* [61] (or another text on type systems, e.g. *TAPL* [99]) for a detailed introductory account of these standard constructs. We will tersely summarize the statics and dynamics of the XL in the next two subsections, respectively.

3.2.3 Statics of the Expanded Language

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

Judgement Form	**Description**
$\Delta \vdash \tau$ type	τ is a type
$\Delta\ \Gamma \vdash e : \tau$	e is assigned type τ

The *type formation judgement*, $\Delta \vdash \tau$ type, is inductively defined by Rules (B.1). The *typing judgement*, $\Delta\ \Gamma \vdash e : \tau$, is inductively defined by Rules (B.2).

Type formation contexts, Δ, are finite sets of hypotheses of the form t type. Empty finite sets are written $\emptyset$, or omitted entirely within judgements, and non-empty finite sets are written as comma-separated finite sequences identified up to exchange and contraction. We write Δ, t type when t type $\notin \Delta$ for Δ extended with the hypothesis t type.

Typing contexts, Γ, are finite functions that map each variable $x \in \text{dom}(\Gamma)$, where $\text{dom}(\Gamma)$ is a finite set of variables, to the hypothesis $x : \tau$, for some τ. Empty typing contexts are written $\emptyset$, or omitted entirely within judgements, and non-empty typing contexts are written as finite sequences of hypotheses identified up to exchange and contraction. We write $\Gamma, x : \tau$, when $x \notin \text{dom}(\Gamma)$, for the extension of Γ with a mapping from x to $x : \tau$, and $\Gamma \cup \Gamma'$ when $\text{dom}(\Gamma) \cap \text{dom}(\Gamma') = \emptyset$ for the typing context mapping each $x \in \text{dom}(\Gamma) \cup \text{dom}(\Gamma')$ to $x : \tau$ if $x : \tau \in \Gamma$ or $x : \tau \in \Gamma'$.

These judgements validate standard lemmas, defined in Appendix B.1: Weakening, Substitution and Decomposition.

3.2.4 Structural Dynamics

The *structural dynamics* (a.k.a. the *structural operational semantics* [101]) of $\mathsf{miniVerse}_{\mathbf{SE}}$ is defined as a transition system by judgements of the following form:

Judgement Form	**Description**
$e \mapsto e'$	e transitions to e'
e val	e is a value

We also define auxiliary judgements for *iterated transition*, $e \mapsto^* e'$, and *evaluation*, $e \Downarrow e'$.

Definition B.6 (Iterated Transition)**.** *Iterated transition, $e \mapsto^* e'$, is the reflexive, transitive closure of the transition judgement, $e \mapsto e'$.*

Definition B.7 (Evaluation)**.** *$e \Downarrow e'$ iff $e \mapsto^* e'$ and e'* val.

Our subsequent developments do not require making reference to particular rules in the structural dynamics (because TSMs operate statically), so we do not reproduce the rules here. Instead, it suffices to state the following conditions.

The Canonical Forms condition characterizes well-typed values. Satisfying this condition requires an *eager* (a.k.a. *by-value*) formulation of the dynamics.

Condition B.8 (Canonical Forms)**.** *If* $\vdash e : \tau$ *and* e val *then:*

1. *If* $\tau = \texttt{parr}(\tau_1;\tau_2)$ *then* $e = \texttt{lam}\{\tau_1\}(x.e')$ *and* $x : \tau_1 \vdash e' : \tau_2$.
2. *If* $\tau = \texttt{all}(t.\tau')$ *then* $e = \texttt{tlam}(t.e')$ *and* t type $\vdash e' : \tau'$.
3. *If* $\tau = \texttt{rec}(t.\tau')$ *then* $e = \texttt{fold}(e')$ *and* $\vdash e' : [\texttt{rec}(t.\tau')/t]\tau'$ *and* e' val.
4. *If* $\tau = \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})$ *then* $e = \texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L})$ *and* $\vdash e_i : \tau_i$ *and* e_i val *for each* $i \in L$.
5. *If* $\tau = \texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i\in L})$ *then for some label set* L' *and label* ℓ *and type* τ'*, we have that* $L = L', \ell$ *and* $\tau = \texttt{sum}[L', \ell](\{i \hookrightarrow \tau_i\}_{i\in L'}; \ell \hookrightarrow \tau')$ *and* $e = \texttt{inj}[\ell](e')$ *and* $\vdash e' : \tau'$ *and* e' val.

The Preservation condition ensures that evaluation preserve typing.

Condition B.9 (Preservation)**.** *If* $\vdash e : \tau$ *and* $e \mapsto^* e'$ *then* $\vdash e' : \tau$.

The Progress condition ensures that evaluating a well-typed expanded expression cannot "get stuck":

Condition B.10 (Progress)**.** *If* $\vdash e : \tau$ *then either* e val *or there exists an* e' *such that* $e \mapsto e'$.

Together, these two conditions constitute the Type Safety Condition.

3.2.5 Syntax of the Unexpanded Language

A $\mathsf{miniVerse}_{\mathbf{SE}}$ program ultimately evaluates as a well-typed expanded expression. However, the programmer does not construct this expanded expression directly. Instead, the programmer constructs an *unexpanded expression*, $\hat{e}$, which might contain *unexpanded types*, $\hat{\tau}$. Figure 3.5 defines the relevant forms.

Unexpanded types and expressions 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 will be technically crucial.

Most of the unexpanded forms in Figure 3.5 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 Appendix B.2.1.

In addition to the stylized syntax given in Figure 3.5, there is also a context-free textual syntax for the UL. Giving a complete definition of the context-free textual syntax as, e.g., a context-free grammar, risks digression into details that are not critical to our purposes here. The paper on Wyvern defines a textual syntax for a similar system [96]. Instead, we need only posit the existence of partial metafunctions $\mathsf{parseUTyp}(b)$ and $\mathsf{parseUExp}(b)$ that go from character sequences, b, to unexpanded types and expressions, respectively.

Condition B.11 (Textual Representability)**.**

1. *For each* $\hat{\tau}$*, there exists* b *such that* $\mathsf{parseUTyp}(b) = \hat{\tau}$.

Sort			Stylized Form	Description
UTyp	$\hat{\tau}$	$::=$	$\hat{t}$	identifier
			$\hat{\tau} \rightharpoonup \hat{\tau}$	partial function
			$\forall \hat{t}.\hat{\tau}$	polymorphic
			$\mu \hat{t}.\hat{\tau}$	recursive
			$\langle \{i \hookrightarrow \hat{\tau}_i\}_{i \in L} \rangle$	labeled product
			$[\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}]$	labeled sum
UExp	$\hat{e}$	$::=$	$\hat{x}$	identifier
			$\hat{e} : \hat{\tau}$	ascription
			$\mathtt{let}\,\mathtt{val}\ \hat{x} = \hat{e}\ \mathtt{in}\ \hat{e}$	value binding
			$\lambda \hat{x}{:}\hat{\tau}.\hat{e}$	abstraction
			$\hat{e}(\hat{e})$	application
			$\Lambda \hat{t}.\hat{e}$	type abstraction
			$\hat{e}[\hat{\tau}]$	type application
			$\mathtt{fold}(\hat{e})$	fold
			$\mathtt{unfold}(\hat{e})$	unfold
			$\langle \{i \hookrightarrow \hat{e}_i\}_{i \in L} \rangle$	labeled tuple
			$\hat{e} \cdot \ell$	projection
			$\mathtt{inj}[\ell](\hat{e})$	injection
			$\mathtt{case}\ \hat{e}\ \{i \hookrightarrow \hat{x}_i.\hat{e}_i\}_{i \in L}$	case analysis
			$\mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{by}\ \mathtt{static}\ e\ \mathtt{in}\ \hat{e}$	seTSM definition
			$\hat{a}$ `` `b` ``	seTSM application

Figure 3.5: Syntax of the $\mathsf{miniVerse}_{\mathbf{SE}}$ unexpanded language (UL).

2. *For each* $\hat{e}$*, there exists* b *such that* $\mathsf{parseUExp}(b) = \hat{e}$*.*

3.2.6 Typed Expansion

Unexpanded expressions, and the unexpanded types therein, are checked and expanded simultaneously according to the *typed expansion judgements*:

Judgement Form	Description
$\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type}$	$\hat{\tau}$ has well-formed expansion τ
$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau$	$\hat{e}$ has expansion e of type τ

Type Expansion

Unexpanded type formation contexts, $\hat{\Delta}$, are of the form $\langle \mathcal{D}; \Delta \rangle$, i.e. they consist of a *type identifier expansion context*, $\mathcal{D}$, paired with a standard type formation context, Δ.

A *type identifier expansion context*, $\mathcal{D}$, is a finite function that maps each type identifier $\hat{t} \in \mathrm{dom}(\mathcal{D})$ to the hypothesis $\hat{t} \rightsquigarrow t$, for some type variable t. We write $\mathcal{D} \uplus \hat{t} \rightsquigarrow t$ for the type identifier expansion context that maps $\hat{t}$ to $\hat{t} \rightsquigarrow t$ and defers to $\mathcal{D}$ for all other type identifiers (i.e. the previous mapping is *updated*.)

We define $\hat{\Delta}, \hat{t} \leadsto t \ \mathsf{type}$ when $\hat{\Delta} = \langle \mathcal{D}; \Delta \rangle$ as an abbreviation of

$$\langle \mathcal{D} \uplus \hat{t} \leadsto t; \Delta, t \ \mathsf{type} \rangle$$

The *type expansion judgement*, $\hat{\Delta} \vdash \hat{\tau} \leadsto \tau \ \mathsf{type}$, is inductively defined by Rules (B.5). The first three of these rules are reproduced below:

$$\frac{}{\hat{\Delta}, \hat{t} \leadsto t \ \mathsf{type} \vdash \hat{t} \leadsto t \ \mathsf{type}} \tag{B.5a}$$

$$\frac{\hat{\Delta} \vdash \hat{\tau}_1 \leadsto \tau_1 \ \mathsf{type} \qquad \hat{\Delta} \vdash \hat{\tau}_2 \leadsto \tau_2 \ \mathsf{type}}{\hat{\Delta} \vdash \hat{\tau}_1 \rightharpoonup \hat{\tau}_2 \leadsto \mathtt{parr}(\tau_1; \tau_2) \ \mathsf{type}} \tag{B.5b}$$

$$\frac{\hat{\Delta}, \hat{t} \leadsto t \ \mathsf{type} \vdash \hat{\tau} \leadsto \tau \ \mathsf{type}}{\hat{\Delta} \vdash \forall \hat{t}.\hat{\tau} \leadsto \mathtt{all}(t.\tau) \ \mathsf{type}} \tag{B.5c}$$

To develop an intuition for how type identifier expansion operates, it is instructive to inspect the derivation of the expansion of the unexpanded type $\forall \hat{t}.\forall \hat{t}.\hat{t}$:

$$\dfrac{\dfrac{\dfrac{}{\langle \hat{t} \leadsto t_2; t_1 \ \mathsf{type}, t_2 \ \mathsf{type} \rangle \vdash \hat{t} \leadsto t_2 \ \mathsf{type}}\ (\text{B.5a})}{\langle \hat{t} \leadsto t_1; t_1 \ \mathsf{type} \rangle \vdash \forall \hat{t}.\hat{t} \leadsto \mathtt{all}(t_2.t_2) \ \mathsf{type}}\ (\text{B.5c})}{\langle \emptyset; \emptyset \rangle \vdash \forall \hat{t}.\forall \hat{t}.\hat{t} \leadsto \mathtt{all}(t_1.\mathtt{all}(t_2.t_2)) \ \mathsf{type}}\ (\text{B.5c})$$

Notice that when Rule (B.5c) is applied, the type identifier expansion context is extended (when the outermost binding is encountered) or updated (at all inner bindings) and the type formation context is simultaneously extended with a (necessarily fresh) hypothesis. Without this mechanism, expansions for unexpanded types with shadowing, like this minimal example, would not exist, because by definition we cannot extend a type formation context with a variable it already mentions, nor implicitly α-vary the unexpanded type to sidestep this problem in the usual manner.

The Type Expansion Lemma establishes that the expansion of an unexpanded type is a well-formed type.

Lemma B.24 (Type Expansion). *If* $\langle \mathcal{D}; \Delta \rangle \vdash \hat{\tau} \leadsto \tau \ \mathsf{type}$ *then* $\Delta \vdash \tau \ \mathsf{type}$.

Proof. By rule induction over Rules (B.5). In each case, we apply the IH to or over each premise, then apply the corresponding type formation rule in Rules (B.1). □

Typed Expression Expansion

Unexpanded typing contexts, $\hat{\Gamma}$, are, similarly, of the form $\langle \mathcal{G}; \Gamma \rangle$, where $\mathcal{G}$ is an *expression identifier expansion context*, and Γ is a 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 $\hat{x} \leadsto x$, for some expression variable, x. We write $\mathcal{G} \uplus \hat{x} \leadsto x$ for the expression identifier expansion context that maps $\hat{x}$ to $\hat{x} \leadsto x$ and defers to $\mathcal{G}$ for all other expression identifiers (i.e. the previous mapping is updated.) We define $\hat{\Gamma}, \hat{x} \leadsto x : \tau$ when $\hat{\Gamma} = \langle \mathcal{G}; \Gamma \rangle$ as an abbreviation of

$$\langle \mathcal{G} \uplus \hat{x} \leadsto x; \Gamma, x : \tau \rangle$$

The *typed expression expansion judgement*, $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau$, is inductively defined by Rules (B.6). Before covering these rules, let us state the main theorem of interest: that typed expansion results in a well-typed expanded expression.

Theorem B.28 (Typed Expression Expansion)**.** *If* $\langle \mathcal{D}; \Delta\rangle\langle \mathcal{G}; \Gamma\rangle \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau$ *then* $\Delta\ \Gamma \vdash e : \tau$*.*

Common Forms Rules (B.6a) through (B.6m) handle unexpanded expressions of common form, as well as ascriptions and let binding. The first five of these rules are reproduced below:

$$\frac{}{\hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi}} \hat{x} \rightsquigarrow x : \tau} \tag{B.6a}$$

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e} : \hat{\tau} \rightsquigarrow e : \tau} \tag{B.6b}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e}_1 \rightsquigarrow e_1 : \tau_1 \qquad \hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau_1 \vdash_{\hat{\Psi}} \hat{e}_2 \rightsquigarrow e_2 : \tau_2}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \mathtt{let\,val}\ \hat{x} = \hat{e}_1\ \mathtt{in}\ \hat{e}_2 \rightsquigarrow \mathtt{ap(lam\{}\tau_1\mathtt{\}(}x.e_2\mathtt{);}e_1\mathtt{)} : \tau_2} \tag{B.6c}$$

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \lambda \hat{x}{:}\hat{\tau}.\hat{e} \rightsquigarrow \mathtt{lam\{}\tau\mathtt{\}(}x.e\mathtt{)} : \mathtt{parr(}\tau\mathtt{;}\tau'\mathtt{)}} \tag{B.6d}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e}_1 \rightsquigarrow e_1 : \mathtt{parr(}\tau\mathtt{;}\tau'\mathtt{)} \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e}_2 \rightsquigarrow e_2 : \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e}_1(\hat{e}_2) \rightsquigarrow \mathtt{ap(}e_1\mathtt{;}e_2\mathtt{)} : \tau'} \tag{B.6e}$$

The rules for the remaining expressions of common form are entirely straightforward, mirroring the corresponding typing rules, i.e. Rules (B.2). The type assigned in the conclusion of each rule is identical to the type assigned in the conclusion of the corresponding typing rule. The seTSM context, $\hat{\Psi}$, passes opaquely through these rules (we will define seTSM contexts below.) As such, the corresponding cases in the proof of Theorem B.28 are by application of the induction hypothesis and the corresponding typing rule.

seTSM Definition and Application The two remaining typed expansion rules, Rules (B.6n) and (B.6o), govern the seTSM definition and application forms. They are defined in the next two subsections, respectively.

3.2.7 seTSM Definitions

The seTSM definition form is

$$\mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{by\ static}\ e_{\text{parse}}\ \mathtt{in}\ \hat{e}$$

An unexpanded expression of this form defines an seTSM identified as $\hat{a}$ with *unexpanded type annotation* $\hat{\tau}$ and *parse function* e_{parse} for use within $\hat{e}$.

Rule (B.6n) defines typed expansion of this form:

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \ \mathsf{type} \qquad \emptyset\,\emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResultSE)} \qquad e_{\text{parse}} \Downarrow e'_{\text{parse}} \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{setsm}(\tau; e'_{\text{parse}})} \hat{e} \rightsquigarrow e : \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{by static}\ e_{\text{parse}}\ \texttt{in}\ \hat{e} \rightsquigarrow e : \tau'} \tag{B.6n}$$

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 the parse function, e_{parse}, is a closed expanded function[3] of the following 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* $b \downarrow_{\mathsf{Body}} e_{\text{body}}$. An inverse mapping is defined by the *body decoding judgement* $e_{\text{body}} \uparrow_{\mathsf{Body}} b$.

Judgement Form	**Description**
$b \downarrow_{\mathsf{Body}} e$	b has encoding e
$e \uparrow_{\mathsf{Body}} b$	e has decoding b

Rather than defining `Body` explicitly, and these judgements inductively against that definition (which would be tedious and uninteresting), it suffices to define the following condition, which establishes an isomorphism between literal bodies and values of type `Body` mediated by the judgements above.

Condition B.16 (Body Isomorphism)**.**

(a) *For every literal body b, we have that $b \downarrow_{\mathsf{Body}} e_{body}$ for some e_{body} such that $\vdash e_{body} : \mathsf{Body}$ and $e_{body}\ \mathsf{val}$.*

(b) *If $\vdash e_{body} : \mathsf{Body}$ and $e_{body}\ \mathsf{val}$ then $e_{body} \uparrow_{\mathsf{Body}} b$ for some b.*

(c) *If $b \downarrow_{\mathsf{Body}} e_{body}$ then $e_{body} \uparrow_{\mathsf{Body}} b$.*

(d) *If $\vdash e_{body} : \mathsf{Body}$ and $e_{body}\ \mathsf{val}$ and $e_{body} \uparrow_{\mathsf{Body}} b$ then $b \downarrow_{\mathsf{Body}} e_{body}$.*

(e) *If $b \downarrow_{\mathsf{Body}} e_{body}$ and $b \downarrow_{\mathsf{Body}} e'_{body}$ then $e_{body} = e'_{body}$.*

(f) *If $\vdash e_{body} : \mathsf{Body}$ and $e_{body}\ \mathsf{val}$ and $e_{body} \uparrow_{\mathsf{Body}} b$ and $e_{body} \uparrow_{\mathsf{Body}} b'$ then $b = b'$.*

The return type of the parse function, `ParseResultSE`, abbreviates a labeled sum type that distinguishes parse errors from successful parses:[4]

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

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

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

[4]In VerseML, the `ParseError` constructor of `parse_result` required an error message and an error location, but we omit these in our formalization for simplicity.

The type abbreviated $\texttt{PrExpr}$ classifies encodings of *proto-expressions*, $\grave{e}$ (pronounced "grave e".) The syntax of proto-expressions, defined in Figure 3.6, will be described when we describe proto-expansion validation in Sec. 3.2.9. The mapping from proto-expressions to values of type $\texttt{PrExpr}$ is defined by the *proto-expression encoding judgement*, $\grave{e} \downarrow_{\mathsf{PrExpr}} e$. An inverse mapping is defined by the *proto-expression decoding judgement*, $e \uparrow_{\mathsf{PrExpr}} \grave{e}$.

Judgement Form	**Description**
$\grave{e} \downarrow_{\mathsf{PrExpr}} e$	$\grave{e}$ has encoding e
$e \uparrow_{\mathsf{PrExpr}} \grave{e}$	e has decoding $\grave{e}$

Again, rather than picking a particular definition of $\texttt{PrExpr}$ and defining the judgements above inductively against it, we only state the following condition, which establishes an isomorphism between values of type $\texttt{PrExpr}$ and proto-expressions.

Condition B.22 (Proto-Expression Isomorphism)**.**

(a) *For every* $\grave{e}$, *we have* $\grave{e} \downarrow_{\mathsf{PrExpr}} e_{proto}$ *for some* e_{proto} *such that* $\vdash e_{proto} : \texttt{PrExpr}$ *and* e_{proto} val.

(b) *If* $\vdash e_{proto} : \texttt{PrExpr}$ *and* e_{proto} val *then* $e_{proto} \uparrow_{\mathsf{PrExpr}} \grave{e}$ *for some* $\grave{e}$.

(c) *If* $\grave{e} \downarrow_{\mathsf{PrExpr}} e_{proto}$ *then* $e_{proto} \uparrow_{\mathsf{PrExpr}} \grave{e}$.

(d) *If* $\vdash e_{proto} : \texttt{PrExpr}$ *and* e_{proto} val *and* $e_{proto} \uparrow_{\mathsf{PrExpr}} \grave{e}$ *then* $\grave{e} \downarrow_{\mathsf{PrExpr}} e_{proto}$.

(e) *If* $\grave{e} \downarrow_{\mathsf{PrExpr}} e_{proto}$ *and* $\grave{e} \downarrow_{\mathsf{PrExpr}} e'_{proto}$ *then* $e_{proto} = e'_{proto}$.

(f) *If* $\vdash e_{proto} : \texttt{PrExpr}$ *and* e_{proto} val *and* $e_{proto} \uparrow_{\mathsf{PrExpr}} \grave{e}$ *and* $e_{proto} \uparrow_{\mathsf{PrExpr}} \grave{e}'$ *then* $\grave{e} = \grave{e}'$.

3. The third premise of Rule (B.6n) evaluates the parse function to a value.

4. The final premise of Rule (B.6n) extends the seTSM context, $\hat{\Psi}$, with the newly determined seTSM definition, and proceeds to assign a type, τ', and expansion, e, to $\hat{e}$. The conclusion of Rule (B.6n) assigns this type and expansion to the seTSM definition as a whole.

   seTSM contexts, $\hat{\Psi}$, are of the form $\langle \mathcal{A}; \Psi \rangle$, where $\mathcal{A}$ is a *TSM identifier expansion context* and Ψ is a *seTSM definition context*.

   A *TSM identifier expansion context*, $\mathcal{A}$, is a finite function mapping each TSM identifier $\hat{a} \in \text{dom}(\mathcal{A})$ to the *TSM identifier expansion*, $\hat{a} \rightsquigarrow a$, for some *TSM name*, a. We write $\mathcal{A} \uplus \hat{a} \rightsquigarrow a$ for the TSM identifier expansion context that maps $\hat{a}$ to $\hat{a} \rightsquigarrow a$, and defers to $\mathcal{A}$ for all other TSM identifiers (i.e. the previous mapping is *updated*.)

   An *seTSM definition context*, Ψ, is a finite function mapping each TSM name $a \in \text{dom}(\Psi)$ to an *expanded seTSM definition*, $a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}})$, where τ is the seTSM's type annotation, and e_{parse} is its parse function. We write $\Psi, a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}})$ when $a \notin \text{dom}(\Psi)$ for the extension of Ψ that maps a to $a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}})$.

   We define $\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}})$, when $\hat{\Psi} = \langle \mathcal{A}; \Psi \rangle$, as an abbreviation of

$$\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}}) \rangle$$

We distinguish TSM identifiers, $\hat{a}$, from TSM names, a, for much the same reason that we distinguish type and expression identifiers from type and expression variables: in order to support TSM definitions identified in the same way as a previously defined TSM definition, without an implicit renaming convention.

3.2.8 seTSM Application

The unexpanded expression form for applying an seTSM named $\hat{a}$ to a literal form with literal body b is:

$$\hat{a}\ \text{`}b\text{`}$$

This stylized form uses backticks to delimit the literal body, but other generalized literal forms, like those described in Figure 3.1, could also be included as derived forms in the textual syntax.

The typed expansion rule governing seTSM application is below:

$$\frac{\begin{array}{c} \hat{\Psi} = \hat{\Psi}', \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{setsm}(\tau; e_{\text{parse}}) \\ b \downarrow_{\mathsf{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \mathtt{inj}[\mathtt{SuccessE}](e_{\text{proto}}) \qquad e_{\text{proto}} \uparrow_{\mathsf{PrExpr}} \grave{e} \\ \mathsf{seg}(\grave{e})\ \mathsf{segments}\ b \qquad \emptyset\, \emptyset \vdash^{\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; b} \grave{e} \rightsquigarrow e : \tau \end{array}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{a}\ \text{`}b\text{`} \rightsquigarrow e : \tau} \tag{B.6o}$$

The premises of Rule (B.6o) 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}. This term is closed per Condition B.16.
3. The third premise applies the parse function e_{parse} to the encoding of the literal body. The parse function is closed by well-formedness of $\hat{\Psi}$ (which, in turn, is maintained by the TSM definition rule, Rule (B.6n), described above).

   If parsing succeeds, i.e. a value of the form $\mathtt{inj}[\mathtt{SuccessE}](e_{\text{proto}})$ results from evaluation, then e_{proto} will be a value of type `PrExpr` (assuming a well-formed seTSM context, by application of the Preservation assumption, Assumption B.9.) We call e_{proto} the *encoding of the proto-expansion*.

   If the parse function produces a value labeled `ParseError`, then typed expansion fails. No rule is necessary to handle this case.
4. The fourth premise decodes the encoding of the proto-expansion to produce the *proto-expansion*, $\grave{e}$, itself.
5. The fifth premise determines a segmentation, $\mathsf{seg}(\grave{e})$, and ensures that it is valid with respect to b. In particular, the predicate ψ $\mathsf{segments}$ b checks that each segment in ψ, has non-negative length and is within bounds of b, and that the segments in ψ do not overlap.

Sort			Operational Form	Stylized Form	Description
PrTyp	$\grave{\tau}$	$::=$	t	t	variable
			$\texttt{prparr}(\grave{\tau};\grave{\tau})$	$\grave{\tau} \rightharpoonup \grave{\tau}$	partial function
			$\texttt{prall}(t.\grave{\tau})$	$\forall t.\grave{\tau}$	polymorphic
			$\texttt{prrec}(t.\grave{\tau})$	$\mu t.\grave{\tau}$	recursive
			$\texttt{prprod}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L})$	$\langle\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}\rangle$	labeled product
			$\texttt{prsum}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L})$	$[\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}]$	labeled sum
			$\texttt{splicedt}[m;n]$	$\texttt{splicedt}[m;n]$	spliced type ref.
PrExp	$\grave{e}$	$::=$	x	x	variable
			$\texttt{prasc}\{\grave{\tau}\}(\grave{e})$	$\grave{e} : \grave{\tau}$	ascription
			$\texttt{prletval}(\grave{e};x.\grave{e})$	$\texttt{let val}\ x = \grave{e}\ \texttt{in}\ \grave{e}$	value binding
			$\texttt{prlam}\{\grave{\tau}\}(x.\grave{e})$	$\lambda x{:}\grave{\tau}.\grave{e}$	abstraction
			$\texttt{prap}(\grave{e};\grave{e})$	$\grave{e}(\grave{e})$	application
			$\texttt{prtlam}(t.\grave{e})$	$\Lambda t.\grave{e}$	type abstraction
			$\texttt{prtap}\{\grave{\tau}\}(\grave{e})$	$\grave{e}[\grave{\tau}]$	type application
			$\texttt{prfold}(\grave{e})$	$\texttt{fold}(\grave{e})$	fold
			$\texttt{prunfold}(\grave{e})$	$\texttt{unfold}(\grave{e})$	unfold
			$\texttt{prtpl}\{L\}(\{i \hookrightarrow \grave{e}_i\}_{i \in L})$	$\langle\{i \hookrightarrow \grave{e}_i\}_{i \in L}\rangle$	labeled tuple
			$\texttt{prprj}[\ell](\grave{e})$	$\grave{e} \cdot \ell$	projection
			$\texttt{prinj}[\ell](\grave{e})$	$\texttt{inj}[\ell](\grave{e})$	injection
			$\texttt{prcase}[L](\grave{e};\{i \hookrightarrow x_i.\grave{e}_i\}_{i \in L})$	$\texttt{case}\ \grave{e}\ \{i \hookrightarrow x_i.\grave{e}_i\}_{i \in L}$	case analysis
			$\texttt{splicede}[m;n;\grave{\tau}]$	$\texttt{splicede}[m;n;\grave{\tau}]$	spliced expr. ref.

Figure 3.6: Syntax of miniVerse$_{\mathbf{SE}}$ proto-types and proto-expressions.

6. The final premise of Rule (B.60) *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.

3.2.9 Syntax of Proto-Expansions

Figure 3.6 defines the syntax of proto-types, $\grave{\tau}$, and proto-expressions, $\grave{e}$. Proto-types and -expressions are ABTs identified up to α-equivalence in the usual manner.

Each expanded form maps onto a proto-expansion form. We refer to these as the *common proto-expansion forms*. The mapping is given explicitly in Appendix B.3.

There are two "interesting" proto-expansion forms, highlighted in yellow in Figure 3.6: a proto-type form for *references to spliced unexpanded types*, $\texttt{splicedt}[m;n]$, and a proto-expression form for *references to spliced unexpanded expressions*, $\texttt{splicede}[m;n;\grave{\tau}]$, where m and n are natural numbers.

3.2.10 Proto-Expansion Validation

The *proto-expansion validation judgements* validate proto-types and proto-expressions and simultaneously generate their final expansions.

Judgement Form	**Description**
$\Delta \vdash^{\mathbb{T}} \grave{\tau} \leadsto \tau \ \mathsf{type}$	$\grave{\tau}$ has well-formed expansion τ
$\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau$	$\grave{e}$ has expansion e of type τ

Type splicing scenes, $\mathbb{T}$, are of the form $\hat{\Delta}; b$ and *expression splicing scenes*, $\mathbb{E}$, are of the form $\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; b$. We write $\mathsf{ts}(\mathbb{E})$ for the type splicing scene constructed by dropping the unexpanded typing context and seTSM context from $\mathbb{E}$:

$$\mathsf{ts}(\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; b) = \hat{\Delta}; b$$

The purpose of splicing scenes is to "remember", during the proto-expansion validation process, the unexpanded type formation context, $\hat{\Delta}$, unexpanded typing context, $\hat{\Gamma}$, seTSM context, $\hat{\Psi}$, and the literal body, b, from the seTSM application site (cf. Rule (B.6o) above.) These structures will be necessary to validate the references to spliced unexpanded types and expressions that appear within the proto-expansion.

Proto-Type Validation

The *proto-type validation judgement*, $\Delta \vdash^{\mathbb{T}} \grave{\tau} \leadsto \tau \ \mathsf{type}$, is inductively defined by Rules (B.9).

Common Forms Rules (B.9a) through (B.9f) validate proto-types of common form. These rules, like the rules for common unexpanded type forms, mirror the corresponding type formation rules, i.e. Rules (B.1). The type splicing scene, $\mathbb{T}$, passes opaquely through these rules. The first three of these are reproduced below.

$$\frac{}{\Delta, t\ \mathsf{type} \vdash^{\mathbb{T}} t \leadsto t\ \mathsf{type}} \quad \text{(B.9a)}$$

$$\frac{\Delta \vdash^{\mathbb{T}} \grave{\tau}_1 \leadsto \tau_1\ \mathsf{type} \qquad \Delta \vdash^{\mathbb{T}} \grave{\tau}_2 \leadsto \tau_2\ \mathsf{type}}{\Delta \vdash^{\mathbb{T}} \mathtt{prparr}(\grave{\tau}_1; \grave{\tau}_2) \leadsto \mathtt{parr}(\tau_1; \tau_2)\ \mathsf{type}} \quad \text{(B.9b)}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash^{\mathbb{T}} \grave{\tau} \leadsto \tau\ \mathsf{type}}{\Delta \vdash^{\mathbb{T}} \mathtt{prall}(t.\grave{\tau}) \leadsto \mathtt{all}(t.\tau)\ \mathsf{type}} \quad \text{(B.9c)}$$

Notice that in Rule (B.9a), only type variables tracked by Δ, the expansion's local type validation context, are well-formed. Type variables tracked by the application site unexpanded type formation context, which is a component of the type splicing scene, $\mathbb{T}$, are not validated.

References to Spliced Types The only proto-type form that does not correspond to a type form is $\texttt{splicedt}[m;n]$, which is a *reference to a spliced unexpanded type*, i.e. it indicates that an unexpanded type should be parsed out from the literal body, which appears in the type splicing scene $\mathbb{T}$, beginning at position m and ending at position n, where m and n are natural numbers. Rule (B.9g) governs this form:

$$\frac{\mathsf{parseUTyp}(\mathsf{subseq}(b;m;n)) = \hat{\tau} \qquad \langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle \vdash \hat{\tau} \leadsto \tau \;\mathsf{type} \qquad \Delta \cap \Delta_{\mathsf{app}} = \emptyset}{\Delta \vdash^{\langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle; b} \texttt{splicedt}[m;n] \leadsto \tau \;\mathsf{type}} \tag{B.9g}$$

The first premise of this rule extracts the indicated subsequence of b using the partial metafunction $\mathsf{subseq}(b;m;n)$ and parses it using the partial metafunction $\mathsf{parseUTyp}(b)$, which was characterized in Sec. 3.2.5, to produce the spliced unexpanded type itself, $\hat{\tau}$.

The second premise of Rule (B.9g) performs type expansion of $\hat{\tau}$ under the application site unexpanded type formation context, $\langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle$, which is a component of the type splicing scene. The hypotheses in the expansion's local type formation context, Δ, are not made available to τ.

The third premise of Rule (B.9g) imposes the constraint that the proto-expansion's type formation context, Δ, be disjoint from the application site type formation context, Δ_{app}. This premise can always be discharged by α-varying the proto-expansion that the reference to the spliced type appears within.

Together, these two premises enforce the injunction on type variable capture as described in Sec. 3.1.5 – the TSM provider can choose type variable names freely within a proto-expansion. We will consider this formally in Sec. 3.2.11 below.

Rules (B.9) validate the following lemma, which establishes that the final expansion of a valid proto-type is a well-formed type under the combined type formation context.

Lemma B.25 (Proto-Expansion Type Validation). *If* $\Delta \vdash^{\langle \mathcal{D}; \Delta_{app} \rangle; b} \grave{\tau} \leadsto \tau \;\mathsf{type}$ *and* $\Delta \cap \Delta_{app} = \emptyset$ *then* $\Delta \cup \Delta_{app} \vdash \tau \;\mathsf{type}$.

Proto-Expression Validation

The *proto-expression validation judgement*, $\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau$, is defined mutually inductively with the typed expansion judgement by Rules (B.10) as follows.

Common Forms Rules (B.10a) through (B.10m) validate proto-expressions of common form, as well as ascriptions and let binding. Once again, the rules for common forms mirror the typing rules, i.e. Rules (B.2). The expression splicing scene, $\mathbb{E}$, passes opaquely through these rules. The first five of these rules are reproduced below:

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash^{\mathbb{E}} x \leadsto x : \tau} \tag{B.10a}$$

$$\frac{\Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau} \leadsto \tau \;\mathsf{type} \qquad \Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \texttt{prasc}\{\grave{\tau}\}(\grave{e}) \leadsto e : \tau} \tag{B.10b}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e}_1 \rightsquigarrow e_1 : \tau_1 \qquad \Delta\ \Gamma, x : \tau_1 \vdash^{\grave{e}_2} e_2 \rightsquigarrow \tau_2 :}{\Delta\ \Gamma \vdash^{\mathbb{E}} \texttt{prletval}(\grave{e}_1; x.\grave{e}_2) \rightsquigarrow \texttt{ap}(\texttt{lam}\{\tau_1\}(x.e_2); e_1) : \tau_2} \tag{B.10c}$$

$$\frac{\Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau} \rightsquigarrow \tau \text{ type} \qquad \Delta\ \Gamma, x : \tau \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \texttt{prlam}\{\grave{\tau}\}(x.\grave{e}) \rightsquigarrow \texttt{lam}\{\tau\}(x.e) : \texttt{parr}(\tau; \tau')} \tag{B.10d}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e}_1 \rightsquigarrow e_1 : \texttt{parr}(\tau; \tau') \qquad \Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e}_2 \rightsquigarrow e_2 : \tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \texttt{prap}(\grave{e}_1; \grave{e}_2) \rightsquigarrow \texttt{ap}(e_1; e_2) : \tau'} \tag{B.10e}$$

Notice that in Rule (B.10a), only variables tracked by the proto-expansion typing context, Γ, are validated. Variables in the application site unexpanded typing context, which appears within the expression splicing scene $\mathbb{E}$, are not validated. This achieves *context independence* as described in Sec. 3.1.5 – seTSMs cannot impose "hidden constraints" on the application site unexpanded typing context, because the variable bindings at the application site are not directly available to proto-expansions. We will consider this formally in Sec. 3.2.11 below.

References to Spliced Unexpanded Expressions The only proto-expression form that does not correspond to an expanded expression form is $\texttt{splicede}[m; n; \grave{\tau}]$, which is a *reference to a spliced unexpanded expression*, i.e. it indicates that an unexpanded expression should be parsed out from the literal body beginning at position m and ending at position n. Rule (B.10n) governs this form:

$$\frac{\begin{array}{c}\emptyset \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau} \rightsquigarrow \tau \text{ type} \qquad \mathbb{E} = \langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle; \langle \mathcal{G}; \Gamma_{\mathsf{app}} \rangle; \hat{\Psi}; b \\ \mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e} \qquad \langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle\ \langle \mathcal{G}; \Gamma_{\mathsf{app}} \rangle \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau \\ \Delta \cap \Delta_{\mathsf{app}} = \emptyset \qquad \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma_{\mathsf{app}}) = \emptyset\end{array}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \texttt{splicede}[m; n; \grave{\tau}] \rightsquigarrow e : \tau} \tag{B.10n}$$

The premises of this rule can be understood as follows:

1. The first premise of this rule validates and expands the type annotation. This type must be context independent.
2. The second premise of this rule serves simply to reveal the components of the expression splicing scene.
3. The third premise of this rule extracts the indicated subsequence of b using the partial metafunction $\mathsf{subseq}(b; m; n)$ and parses it using the partial metafunction $\mathsf{parseUExp}(b)$, characterized in Sec. 3.2.5, to produce the referenced spliced unexpanded expression, $\hat{e}$.
4. The fourth premise of Rule (B.10n) performs typed expansion of $\hat{e}$ assuming the application site contexts that appear in the expression splicing scene. Notice that the hypotheses in Δ and Γ are not made available to $\hat{e}$.
5. The fifth premise of Rule (B.10n) imposes the constraint that the proto-expansion's type formation context, Δ, be disjoint from the application site type formation context, Δ_{app}. Similarly, the sixth premise requires that the proto-expansion's

typing context, Γ, be disjoint from the application site typing context, Γ_{app}. These two premises can always be discharged by α-varying the proto-expression that the reference to the spliced unexpanded expression appears within. Together, these premises enforce the prohibition on capture as described in Sec. 3.1.5 – the TSM provider can choose variable names freely within a proto-expansion, because the language prevents them from shadowing those at the application site. Again, we will consider this formally in Sec. 3.2.11 below.

3.2.11 Metatheory

Typed Expansion

Let us now consider Theorem B.28, which was mentioned at the beginnning of Sec. 3.2.6 and is reproduced below:

Theorem B.28 (Typed Expression Expansion)**.** *If* $\langle \mathcal{D}; \Delta \rangle \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau$ *then* $\Delta\ \Gamma \vdash e : \tau$.

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:

Theorem B.27 (Typed Expansion (Full))**.**

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* $\Delta \cup \Delta_{app}\ \Gamma \cup \Gamma_{app} \vdash e : \tau$.

Proof. By mutual rule induction over Rules (B.6) and Rules (B.10). The full proof is given in Appendix B.4.3. We will reproduce the interesting cases below.

The proof of part 1 proceeds by inducting over the typed expansion assumption. The only interesting cases are those related to seTSM definition and application, reproduced below. In the following cases, let $\hat{\Delta} = \langle \mathcal{D}; \Delta \rangle$ and $\hat{\Gamma} = \langle \mathcal{G}; \Gamma \rangle$ and $\hat{\Psi} = \langle \mathcal{A}; \Psi \rangle$.

Case (B.6n). We have

(1) $\hat{e} = \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}'\ \mathtt{by}\ \mathtt{static}\ e_{\text{parse}}\ \mathtt{in}\ \hat{e}'$	by assumption
(2) $\hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'\ \mathsf{type}$	by assumption
(3) $\emptyset\ \emptyset \vdash e_{\text{parse}} : \mathtt{parr}(\mathtt{Body}; \mathtt{ParseResultSE})$	by assumption
(4) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{setsm}(\tau'; e_{\text{parse}})} \hat{e}' \rightsquigarrow e : \tau$	by assumption
(5) $\Delta \vdash \tau'\ \mathsf{type}$	by Lemma B.24 to (2)
(6) $\Delta\ \Gamma \vdash e : \tau$	by IH, part 1(a) on (4)

Case (B.6o). We have

(1) $\hat{e} = \hat{a}\ \text{‘}b\text{‘}$	by assumption
(2) $\mathcal{A} = \mathcal{A}', \hat{a} \rightsquigarrow a$	by assumption
(3) $\Psi = \Psi', a \hookrightarrow \mathtt{setsm}(\tau; e_{\text{parse}})$	by assumption
(4) $b \downarrow_{\mathsf{Body}} e_{\text{body}}$	by assumption
(5) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \mathtt{inj}[\mathtt{SuccessE}](e_{\text{proto}})$	by assumption
(6) $e_{\text{proto}} \uparrow_{\mathsf{PrExpr}} \grave{e}$	by assumption

(7) $\emptyset\,\emptyset \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};b} \grave{e} \rightsquigarrow e : \tau$ — by assumption

(8) $\emptyset \cap \Delta = \emptyset$ — by finite set intersection

(9) $\emptyset \cap \mathrm{dom}(\Gamma) = \emptyset$ — by finite set intersection

(10) $\emptyset \cup \Delta\ \emptyset \cup \Gamma \vdash e : \tau$ — by IH, part 2 on (7), (8), and (9)

(11) $\Delta\ \Gamma \vdash e : \tau$ — by finite set and finite function identity over (10)

The proof of part 2 proceeds by induction over the proto-expression validation assumption. The only interesting case governs references to spliced expressions. In the following cases, let $\hat{\Delta}_{\text{app}} = \langle \mathcal{D}; \Delta_{\text{app}} \rangle$ and $\hat{\Gamma}_{\text{app}} = \langle \mathcal{G}; \Gamma_{\text{app}} \rangle$ and $\hat{\Psi} = \langle \mathcal{A}; \Psi \rangle$.

Case (B.10n)**.**

(1) $\grave{e} = \mathtt{splicede}[m; n; \grave{\tau}]$ — by assumption

(2) $\mathbb{E} = \langle \mathcal{D}; \Delta_{\text{app}} \rangle;\ \langle \mathcal{G}; \Gamma_{\text{app}} \rangle;\ \hat{\Psi};\ b$ — by assumption

(3) $\emptyset \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau} \rightsquigarrow \tau\ \mathsf{type}$ — by assumption

(4) $\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e}$ — by assumption

(5) $\hat{\Delta}_{\text{app}}\ \hat{\Gamma}_{\text{app}} \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau$ — by assumption

(6) $\Delta \cap \Delta_{\text{app}} = \emptyset$ — by assumption

(7) $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma_{\text{app}}) = \emptyset$ — by assumption

(8) $\Delta_{\text{app}}\ \Gamma_{\text{app}} \vdash e : \tau$ — by IH, part 1 on (5)

(9) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash e : \tau$ — by Lemma B.2 over Δ and Γ and exchange on (8)

The mutual induction can be shown to be well-founded by showing that the following numeric metric on the judgements that we induct over is decreasing:

$$\|\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau\| = \|\hat{e}\|$$
$$\|\Delta\ \Gamma \vdash^{\hat{\Delta}_{\text{app}};\hat{\Gamma}_{\text{app}};\hat{\Psi};b} \grave{e} \rightsquigarrow e : \tau\| = \|b\|$$

where $\|b\|$ is the length of b and $\|\hat{e}\|$ is the sum of the lengths of the literal bodies in $\hat{e}$ (see Appendix B.2.1.)

The only case in the proof of part 1 that invokes part 2 is Case (B.6o). There, we have that the metric remains stable:

$$\begin{aligned} &\|\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{a}\ \text{‘}b\text{‘} \rightsquigarrow e : \tau\| \\ =&\|\emptyset\ \emptyset \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};b} \grave{e} \rightsquigarrow e : \tau\| \\ =&\|b\| \end{aligned}$$

The only case in the proof of part 2 that invokes part 1 is Case (B.10n). There, we have that $\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e}$ and the IH is applied to the judgement $\hat{\Delta}_{\text{app}}\ \hat{\Gamma}_{\text{app}} \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau$ where $\hat{\Delta}_{\text{app}} = \langle \mathcal{D}; \Delta_{\text{app}} \rangle$ and $\hat{\Gamma}_{\text{app}} = \langle \mathcal{G}; \Gamma_{\text{app}} \rangle$ and $\hat{\Psi} = \langle \mathcal{A}; \Psi \rangle$.

Because the metric is stable when passing from part 1 to part 2, we must have that it is strictly decreasing in the other direction:

$$\|\hat{\Delta}_{\text{app}}\ \hat{\Gamma}_{\text{app}} \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau\| < \|\Delta\ \Gamma \vdash^{\hat{\Delta}_{\text{app}};\hat{\Gamma}_{\text{app}};\hat{\Psi};b} \texttt{splicede}[m;n;\grave{\tau}] \rightsquigarrow e : \tau\|$$

i.e. by the definitions above,

$$\|\hat{e}\| < \|b\|$$

This is established by appeal to the following two conditions. The first condition states that an unexpanded expression constructed by parsing a textual sequence b is strictly smaller, as measured by the metric defined above, than the length of b, because some characters must necessarily be used to invoke a TSM and delimit each literal body.

Condition B.12 (Expression Parsing Monotonicity)**.** *If* $\mathsf{parseUExp}(b) = \hat{e}$ *then* $\|\hat{e}\| < \|b\|$*.*

The second condition simply states that subsequences of b are no longer than b.

Condition B.17 (Body Subsequencing)**.** *If* $\mathsf{subseq}(b; m; n) = b'$ *then* $\|b'\| \leq \|b\|$*.*

Combining these two conditions, we have that $\|\hat{e}\| < \|b\|$ as needed. □

Abstract Reasoning Principles

The following theorem summarizes the abstract reasoning principles that programmers can rely on when applying an seTSM. A description of each named clause is given in-line below.

Theorem B.31 (seTSM 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)*** $\hat{\Psi} = \hat{\Psi}', \hat{a} \rightsquigarrow a \hookrightarrow \texttt{setsm}(\tau; e_{parse})$ *and* $\Delta\ \Gamma \vdash e : \tau$

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

2. $b \downarrow_{\mathsf{Body}} e_{body}$
3. $e_{parse}(e_{body}) \Downarrow \texttt{inj}[\texttt{SuccessE}](e_{proto})$
4. $e_{proto} \uparrow_{\mathsf{PrExpr}} \grave{e}$
5. ***(Segmentation)*** $\mathsf{seg}(\grave{e})$ segments b

   The segmentation determined by the proto-expansion actually segments the literal body (i.e. each segment is in-bounds and the segments are non-overlapping.)

6. $\mathsf{summary}(\grave{e}) = \{\texttt{splicedt}[m'_i; n'_i]\}_{0 \leq i < n_{ty}} \cup \{\texttt{splicede}[m_i; n_i; \grave{\tau}_i]\}_{0 \leq i < n_{exp}}$
7. ***(Typing 2)*** $\{\langle\mathcal{D};\Delta\rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow \tau'_i\ \mathsf{type}\}_{0 \leq i < n_{ty}}$ *and* $\{\Delta \vdash \tau'_i\ \mathsf{type}\}_{0 \leq i < n_{ty}}$

   Each spliced type has a well-formed expansion at the application site.

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

   Each type annotation on a reference to a spliced expression has a well-formed expansion at the application site.

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

   Each spliced expression has a well-typed expansion consistent with its type annotation.

10. *(**Capture Avoidance**) $e = [\{\tau'_i/t_i\}_{0\leq i<n_{ty}}, \{e_i/x_i\}_{0\leq i<n_{exp}}]e'$ for some $\{t_i\}_{0\leq i<n_{ty}}$ and $\{x_i\}_{0\leq i<n_{exp}}$ and e'*

    The final expansion can be decomposed into a term with variables in place of each spliced type or expression. The expansions of these spliced types and expressions can be substituted into this term in the standard capture avoiding manner.

11. *(**Context Independence**) $\mathsf{fv}(e') \subset \{t_i\}_{0\leq i<n_{ty}} \cup \{x_i\}_{0\leq i<n_{exp}}$*

    The aforementioned decomposed term makes no mention of bindings in the application site context.

Proof. The proof, which involves auxiliary lemmas about the decomposition of proto-types and proto-expressions, is given in Appendix B.4.4. □

This style of specifying the hygiene properties builds directly on the standard notion of capture-avoiding substitution for general ABTs. Prior work on hygiene for macro systems has instead explicitly specified how fresh variables are generated during expansion (e.g. [66].) Our formal approach appears therefore to be more elegant in this regard.

Chapter 4

Simple Pattern TSMs (spTSMs)

In Chapter 3, our interest was in situations where the programmer needed to *construct* (a.k.a. *introduce*) a value. In this chapter, we consider situations where the programmer needs to *deconstruct* (a.k.a. *eliminate*) a value by pattern matching. For example, consider again the recursive labeled sum type `rx` defined in Figure 2.2. We can pattern match over a value `r` of type `rx` using VerseML's **`match`** construct as shown in the example below:

```
1 fun is_seq(r : rx) =>
2   match r with
3     Seq(Str(name), Seq(Str ": ", ssn)) => Some (name, ssn)
4   | _ => None
5   end
```

Match expressions consist of a *scrutinee*, here `r`, and a sequence of *rules* separated by vertical bars, `|`, in the textual syntax. Each rule consists of a *pattern* and an expression called the corresponding *branch*, separated by a double arrow, `=>`, in the textual syntax. During evaluation, the value of the scrutinee is matched against each pattern sequentially. If a match occurs, evaluation proceeds along the corresponding branch.

A variable can appear at most once in a valid pattern. In the corresponding branch, the variable stands for the value it matched. For example, on Line 3 above, the pattern

```
Seq(Str(name), Seq(Str ": ", ssn))
```

matches values with the following structure:

`Seq(Str(`e_1`), Seq(Str ": ", `e_2`))`

where e_1 is a value of type `string` and e_2 is a value of type `rx`. The variables `name` and `ssn` stand for the values of e_1 and e_2, respectively, in the corresponding branch expression.

On Line 4 above, the pattern `_` is the *wildcard pattern* – it matches any value, like the variable pattern, but binds no variables.

The behavior of the **`match`** construct when no pattern in the rule sequence matches a value is to raise an exception indicating *match failure*. It is possible to statically determine whether match failure is possible (i.e. whether there exist values of the scrutinee that do not match any pattern in the rule sequence.) A rule sequence that cannot lead to match failure is said to be *exhaustive*. Compilers warn the programmer when a rule sequence is non-exhaustive. In the example above, our use of the wildcard pattern ensures that match failure cannot occur.

It is also possible to statically decide when a rule is *redundant* relative to the preceding rules. For example, if we add another rule at the end of the match expression above, it will be redundant because all values match the wildcard pattern. Again, compilers warn the programmer when a rule is redundant.

Nested pattern matching generalizes the projection and case analysis operators (i.e. the *eliminators*) for products and sums (cf. miniVerse$_{\mathbf{SE}}$ from the previous section.)

In Sec. 2.3.2, we considered a hypothetical dialect of VerseML called $\mathcal{V}_{\texttt{rx}}$ with derived regex pattern forms. In this dialect, we can express the example above at lower syntactic cost using standard POSIX regex syntax extended with pattern splicing forms:

```
fun f(r : rx) =>
  match r with
    /@name: %ssn/ => Some (name, ssn)
  | _ => None
  end
```

This dialect-oriented approach has problems, as discussed in Chapter 2.4.8.

Expression TSMs – introduced in Chapter 3 – can decrease the syntactic cost of constructing a value, but expressions are syntactically and semantically distinct from patterns, so we cannot simply apply an expression TSM within a pattern.[1] We need a new (albeit closely related) construct – the **pattern TSM**. In this chapter, we consider only **simple pattern TSMs** (spTSMs), i.e. pattern TSMs that generate patterns that match values of a single specified type, like `rx`. In Chapter 5, we will consider both expression and pattern TSMs that specify type and module parameters (peTSMs and ppTSMs).

The organization of the remainder of this chapter mirrors that of Chapter 3. We begin in Sec. 4.1 with a "tutorial-style" introduction to spTSMs in VerseML. Then, in Sec. 4.2, we define an extension of miniVerse$_{\mathbf{SE}}$ called miniVerse$_{\mathbf{S}}$ that makes the intuitions developed in Sec. 4.1 mathematically precise.

4.1 Simple Pattern TSMs By Example

4.1.1 Usage

The VerseML function `f` defined at the beginning of this chapter can be expressed at lower syntactic cost by applying an spTSM named `$rx` as follows:

```
fun f(r : rx) =>
  match r with
    $rx /@name: %ssn/ => Some (name, ssn)
  | _ => None
  end
```

[1]The fact that certain concrete expression and pattern forms coincidentally overlap is immaterial to this fundamental distinction.

Like expression TSMs, pattern TSMs are applied to *generalized literal forms* (see Figure 3.1.) During the *typed expansion* phase, the applied pattern TSM parses the body of the literal form to generate a *proto-expansion*. The language validates the proto-expansion according to criteria that we will establish in Sec. 4.1.5. If validation succeeds, the language generates the final expansion (or more concisely, simply the expansion) of the pattern. The expansion of the unexpanded pattern `$rx /@name: %ssn/` from the example above is the following pattern:

```
Seq(Str(name), Seq(Str ": ", ssn))
```

The checks for exhaustiveness and redundancy are performed post-expansion.

For convenience, the programmer can specify a TSM at the outset of a sequence of rules that is applied to every outermost generalized literal form. For example, the function `is_dna_rx` from Figure 2.3 and Figure 2.10 can be expressed using the spTSM `$rx` as follows:

```
fun is_dna_rx(r : rx) : boolean =>
  match r using $rx with
  | /A/ => True
  | /T/ => True
  | /G/ => True
  | /C/ => True
  | /%(r1)%(r2)/ => (is_dna_rx r1) andalso (is_dna_rx r2)
  | /%(r1)|%(r2)/ => (is_dna_rx r1) andalso (is_dna_rx r2)
  | /%(r)*/ => is_dna_rx r'
  | _ => False
  end
```

4.1.2 Definition

The definition of the pattern TSM `$rx` shown being applied in the examples above takes the following form:

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

This definition first names the pattern TSM `$rx`. Pattern TSM names, like expression TSM names, must begin with the dollar sign (`$`) to distinguish them from labels. Pattern TSM names and expression TSM names are tracked separately, i.e. an expression TSM and a pattern TSM can have the same name without conflict (as is the case here – the expression TSM that was described in Sec. 3.1.2 is also named `$rx`.)

The *sort qualifier* **`for patterns`** indicates that this is a pattern TSM definition, rather than an expression TSM definition (the sort qualifier **`for expressions`** can be written for expression TSMs, though when the sort qualifier is omitted this is the default.) Defining both an expression TSM and a pattern TSM with the same name at the same type is a common idiom, so VerseML defines a derived form for combining their definitions:

```
type proto_pat = rec(proto_pat =>
                   (* no variable pattern form *)
                   Wild
                 + (* ... *)
                 + SplicedP of segment * proto_typ)
```

Figure 4.1: Abbreviated definition of `proto_pat` in the VerseML prelude.

```
syntax $rx at rx for expressions by
  static fn(body : body) : parse_result(proto_expr) =>
    (* regex expression parser here *)
for patterns by
  static fn(body : body) : parse_result(proto_pat) =>
    (* regex pattern parser here *)
end
```

Pattern TSMs, like expression TSMs, must specify a static parse function. For pattern TSMs, the parse function must be of type `body -> parse_result(proto_pat)`, where `body` and `parse_result` are defined as in Figure 3.2.

The type `proto_pat`, defined in Figure 4.1, is analagous to the types `proto_expr` and `proto_typ` defined in Figure 3.3. This type classifies *encodings of proto-patterns*. Every pattern form has a corresponding proto-pattern form, with the exception of variable patterns (for reasons explained in Sec. 4.1.5 below.) There is also an additional constructor, `SplicedP`, to allow a proto-pattern to refer indirectly to spliced patterns by their location within the literal body.

4.1.3 Splicing

Spliced patterns are unexpanded patterns that appear directly within the literal body of another unexpanded pattern. For example, `name` and `ssn` appear within the unexpanded pattern `$rx /@name: %ssn/`. When the parse function determines that a subsequence of the literal body should be taken as a spliced pattern (here, by recognizing the characters `@` or `%` followed by a variable or parenthesized pattern), it can refer to it within the proto-expansion that it computes using the `SplicedP` variant of the `proto_pat` type shown in Figure 4.1. This variant takes a value of type `segment` because proto-patterns refer to spliced patterns indirectly by their position within the literal body. This prevents pattern TSMs from "forging" a spliced pattern (i.e. claiming that some pattern is a spliced pattern, even though it does not appear in the literal body.) Like references to spliced expressions, each reference to a spliced pattern must also specify a type.

The proto-expansion generated by the pattern TSM `$rx` for the example above, if written in a hypothetical concrete syntax where references to spliced patterns are written **`spliced`**`<startIdx; endIdx; ty>`, is:

```
Seq(Str(spliced<1; 4; string>),
    Seq(Str ": ", spliced<8; 10; rx>))
```

Here, `spliced<1; 4; string>` refers to the string subpattern `name` by location, and similarly, `spliced<8; 10; rx>` refers to the regex subpattern `ssn` by location.

4.1.4 Splice Summaries and Segmentations

The *splice summary* of a proto-pattern is the finite set of references to spliced types or patterns. The *segmentation* of a proto-pattern is the finite set of locations in the splice summary. For example, the segmentation of the literal body is the following finite set:

$$\{(1,4),(8,10)\}$$

As with references to spliced expressions, the language checks that the references to spliced terms in a proto-expansion are 1) within bounds of the literal body and 2) non-overlapping.

4.1.5 Proto-Expansion Validation

After the pattern TSM generates a proto-expansion, the language must validate it to generate a final expansion. This also serves to maintain a reasonable type and binding discipline.

Typing

To maintain a reasonable type discipline, proto-expansion validation checks:

1. that each spliced pattern matches values of the type indicated in the summary; and
2. that the final expansion matches values of the type specified in the type annotation on the pattern TSM definition, e.g. the type `rx` above.

Hidden Bindings

To maintain a useful binding discipline, i.e. to allow programmers to reason about variable binding without examining TSM expansions directly, the validation process allows variable patterns to occur only in spliced patterns (just as variables bound at the use site can only appear in spliced expressions when using an expression TSM.) Indeed, there is no constructor for the type `proto_pat` corresponding to a variable pattern. This prohibition on "hidden bindings" is beneficial because the client can rely on the fact that no variables other than those that appear directly within the pattern at the application site are bound in the corresponding branch expression. This prohibition on hidden bindings is analagous to the prohibition on capture discussed in Sec. 3.1.5 (differing in that it is concerned with the bindings visible to the corresponding branch expression, rather than to spliced expressions.)

4.1.6 Final Expansion

If validation succeeds, the semantics generates the *final expansion* of the pattern where the references to spliced patterns in the proto-pattern have been replaced by their respective final expansions. For example, the final expansion of `$rx /@name: %ssn/` is:

```
Seq(Str(name), Seq(Str ": ", ssn))
```

4.2 miniVerse$_{\mathbf{S}}$

To make the intuitions developed in the previous section about pattern TSMs precise, we now introduce miniVerse$_{\mathbf{S}}$, a reduced dialect of VerseML with support for both seTSMs and spTSMs. Like miniVerse$_{\mathbf{SE}}$, miniVerse$_{\mathbf{S}}$ consists of an *unexpanded language* (UL) defined by typed expansion to a standard *expanded language* (XL). The full definition of miniVerse$_{\mathbf{S}}$ is given in Appendix B superimposed upon the definition of miniVerse$_{\mathbf{SE}}$. We will focus on the rules specifically related to pattern matching and spTSMs below.

Our formulation of pattern matching is adapted from Harper's formulation in *Practical Foundations for Programming Languages, First Edition* [60].

4.2.1 Syntax of the Expanded Language

Figure 4.2.1 defines the syntax of the miniVerse$_{\mathbf{S}}$ *expanded language* (XL), which consists of *types*, τ, *expanded expressions*, e, *expanded rules*, r, and *expanded patterns*, p. The miniVerse$_{\mathbf{S}}$ XL differs from the miniVerse$_{\mathbf{SE}}$ XL only by the addition of the pattern matching operator and related forms.[2]

The main syntactic feature of note is that the rule form places a pattern, p, in the binder position:

$$\texttt{rule}(p.e)$$

This can be understood as binding all of the variables in p for use within e. A small technical note: the ABT *renaming* meta-operation (which underlies the notion of alpha-equivalence) requires that these variables appear as a sequence. Rather than redefining this metaoperation explicitly, we implicitly determine such a sequence by performing a depth-first traversal, with traversal of the labeled tuple pattern form, $\texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L})$, relying on some (arbitrary) total ordering on labels.

[2]The projection and case analysis operators can be defined in terms of the match operator, but to simplify the appendix, we leave them in place.

Sort			Operational Form	Description
Typ	τ	$::=$	$\cdots$	(see Figure 3.4)
Exp	e	$::=$	$\cdots$	(see Figure 3.4)
			$\texttt{match}[n](e; \{r_i\}_{1 \le i \le n})$	match
Rule	r	$::=$	$\texttt{rule}(p.e)$	rule
Pat	p	$::=$	x	variable pattern
			$\texttt{wildp}$	wildcard pattern
			$\texttt{foldp}(p)$	fold pattern
			$\texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L})$	labeled tuple pattern
			$\texttt{injp}[\ell](p)$	injection pattern

Figure 4.2: Syntax of the miniVerse$_\mathbf{S}$ expanded language (XL)

4.2.2 Statics of the Expanded Language

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

Judgement Form	Description
$\Delta \vdash \tau$ type	τ is a well-formed type
$\Delta\, \Gamma \vdash e : \tau$	e is assigned type τ
$\Delta\, \Gamma \vdash r : \tau \Mapsto \tau'$	r takes values of type τ to values of type τ'
$\Delta \vdash p : \tau \dashV \Gamma$	p matches values of type τ and generates hypotheses Γ

The types of miniVerse$_\mathbf{S}$ are exactly those of miniVerse$_\mathbf{SE}$, described in Sec. 3.2, so the *type formation judgement*, $\Delta \vdash \tau$ type, is inductively defined by Rules (B.1) as before.

The *typing judgement*, $\Delta\, \Gamma \vdash e : \tau$, assigns types to expressions and is inductively defined by Rules (B.2), which consist of:

- The typing rules of miniVerse$_\mathbf{SE}$, i.e. Rules (B.2a) through (B.2k).
- The following rule for match expressions:

$$\frac{\Delta\, \Gamma \vdash e : \tau \qquad \{\Delta\, \Gamma \vdash r_i : \tau \Mapsto \tau'\}_{1 \le i \le n}}{\Delta\, \Gamma \vdash \texttt{match}[n](e; \{r_i\}_{1 \le i \le n}) : \tau'} \tag{B.2l}$$

The first premise of Rule (B.2l) assigns a type, τ, to the scrutinee, e. The second premise then ensures that each rule r_i, for $1 \le i \le n$, takes values of type τ to values of the type of the match expression as a whole, τ', according to the *rule typing judgement*, $\Delta\, \Gamma \vdash r : \tau \Mapsto \tau'$, which is defined mutually with Rules (B.2) by the following rule:

$$\frac{\Delta \vdash p : \tau \dashV \Gamma' \qquad \Delta\, \Gamma \cup \Gamma' \vdash e : \tau'}{\Delta\, \Gamma \vdash \texttt{rule}(p.e) : \tau \Mapsto \tau'} \tag{B.3}$$

The first premise invokes the *pattern typing judgement*, $\Delta \vdash p : \tau \dashV \Gamma'$, to check that the pattern, p, matches values of type τ (defined assuming Δ), and to gather the typing hypotheses that the pattern generates in a typing context, Γ'. (Algorithmically, the typing context is the "output" of the pattern typing judgement.) The second premise of Rule

(B.3) extends the incoming typing context, Γ, with the hypotheses generated by pattern typing, Γ, and checks the branch expression, e, against the branch type, τ'.

The pattern typing judgement is inductively defined by Rules (B.4). Rule (B.4a) specifies that a variable pattern, x, matches values of any type, τ, and generates the hypothesis that x has type τ:

$$\frac{}{\Delta \vdash x : \tau \dashv\!\mid x : \tau} \tag{B.4a}$$

Rule (B.4b) specifies that a wildcard pattern also matches values of any type, τ, but wildcard patterns generate no hypotheses:

$$\frac{}{\Delta \vdash \texttt{wildp} : \tau \dashv\!\mid \emptyset} \tag{B.4b}$$

Rule (B.4c) specifies that a fold pattern, $\texttt{foldp}(p)$, matches values of the recursive type $\texttt{rec}(t.\tau)$ if p matches values of a single unrolling of the recursive type, $[\texttt{rec}(t.\tau)/t]\tau$:

$$\frac{\Delta \vdash p : [\texttt{rec}(t.\tau)/t]\tau \dashv\!\mid \Gamma}{\Delta \vdash \texttt{foldp}(p) : \texttt{rec}(t.\tau) \dashv\!\mid \Gamma} \tag{B.4c}$$

Rule (B.4d) specifies that a labeled tuple pattern matches values of the labeled product type $\texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})$. Labeled tuple patterns, $\texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L})$, specify a subpattern p_i for each label $i \in L$. The premise checks each subpattern p_i against the corresponding type τ_i, generating hypotheses Γ_i. The conclusion of the rule gathers these hypotheses into a single pattern typing context, $\cup_{i \in L} \Gamma_i$:

$$\frac{\{\Delta \vdash p_i : \tau_i \dashv\!\mid \Gamma_i\}_{i \in L}}{\Delta \vdash \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\!\mid \cup_{i \in L} \Gamma_i} \tag{B.4d}$$

The definition of typing context extension, applied iteratively here, implicitly requires that the pattern typing contexts Γ_i be mutually disjoint, i.e.

$$\{\{\text{dom}(\Gamma_i) \cap \text{dom}(\Gamma_j) = \emptyset\}_{j \in L \setminus i}\}_{i \in L}$$

Finally, Rule (B.4e) specifies that an injection pattern, $\texttt{injp}[\ell](p)$, matches values of labeled sum types of the form $\texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)$, i.e. labeled sum types that define a case for the label ℓ. The pattern p must match value of type τ and generate hypotheses Γ:

$$\frac{\Delta \vdash p : \tau \dashv\!\mid \Gamma}{\Delta \vdash \texttt{injp}[\ell](p) : \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\!\mid \Gamma} \tag{B.4e}$$

4.2.3 Structural Dynamics

The *structural dynamics of* $\mathsf{miniVerse}_\mathsf{S}$ is defined as a transition system, and is organized around judgements of the following form:

Judgement Form	**Description**
$e \mapsto e'$	e transitions to e'
e val	e is a value
e matchfail	e raises match failure

We also define auxiliary judgements for *iterated transition*, $e \mapsto^* e'$, and *evaluation*, $e \Downarrow e'$.

Definition B.6 (Iterated Transition). *Iterated transition, $e \mapsto^* e'$, is the reflexive, transitive closure of the transition judgement, $e \mapsto e'$.*

Definition B.7 (Evaluation). *$e \Downarrow e'$ iff $e \mapsto^* e'$ and e' val.*

As in Sec. 3.2.4, our subsequent developments do not make mention of particular rules in the dynamics, nor do they make mention of other judgements, not listed above, that are used only for defining the dynamics of the match operator, so we do not produce these details here. Instead, it suffices to state the following conditions.

The Canonical Forms condition, which characterizes well-typed values, is identical to the corresponding condition in the structural dynamics of $\mathsf{miniVerse}_{\mathbf{SE}}$, i.e. Condition B.8.

The Preservation condition ensures that evaluation preserves typing, and is again identical to the corresponding condition in the structural dynamics of $\mathsf{miniVerse}_{\mathbf{SE}}$.

Condition B.9 (Preservation). *If $\vdash e : \tau$ and $e \mapsto e'$ then $\vdash e' : \tau$.*

The Progress condition ensures that evaluation of a well-typed expanded expression cannot "get stuck". We must consider the possibility of match failure in this condition.

Condition B.10 (Progress). *If $\vdash e : \tau$ then either e val or e $\mathsf{matchfail}$ or there exists an e' such that $e \mapsto e'$.*

Together, these two conditions constitute the Type Safety Condition.

We do not define the semantics of exhaustiveness and redundancy checking here, because these can be checked post-expansion (but see [60] for a formal account.)

4.2.4 Syntax of the Unexpanded Language

The syntax of the $\mathsf{miniVerse}_{\mathbf{S}}$ unexpanded language (UL) extends the syntax of the $\mathsf{miniVerse}_{\mathbf{SE}}$ unexpanded language as shown in Figure 4.3.

Sort			**Stylized Form**	**Description**
UTyp	$\hat{\tau}$	$::=$	$\cdots$	(see Figure 3.5)
UExp	$\hat{e}$	$::=$	$\cdots$	(see Figure 3.5)
			$\mathtt{match}\ \hat{e}\ \{\hat{r}_i\}_{1 \leq i \leq n}$	match
			$\mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{for\ patterns\ by\ static}\ e\ \mathtt{in}\ \hat{e}$	spTSM definition
URule	$\hat{r}$	$::=$	$\hat{p} \Rightarrow \hat{e}$	match rule
UPat	$\hat{p}$	$::=$	$\hat{x}$	identifier pattern
			$\_$	wildcard pattern
			$\mathtt{fold}(\hat{p})$	fold pattern
			$\langle\{i \hookrightarrow \hat{p}_i\}_{i \in L}\rangle$	labeled tuple pattern
			$\mathtt{inj}[\ell](\hat{p})$	injection pattern
			$\hat{a}\ \text{`}b\text{`}$	spTSM application

Figure 4.3: Syntax of the $\mathsf{miniVerse}_{\mathbf{S}}$ unexpanded language

As in $\mathsf{miniVerse}_{\mathbf{SE}}$, each expanded form has a corresponding unexpanded form. We refer to these as the *common forms*. The correspondence is defined in Appendix B.2.1. There are two forms related specifically to spTSMs, highlighted in yellow above: the spTSM definition form and the spTSM application form.

In addition to the stylized syntax given in Figure 3.5, there is also a context-free textual syntax for the UL. Again, 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 B.11 (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* $\mathsf{parseUPat}(b) = \hat{p}$*.*

4.2.5 Typed Expansion

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

Judgement Form	**Description**
$\hat{\Delta} \vdash \hat{\tau} \leadsto \tau\ \mathsf{type}$	$\hat{\tau}$ has well-formed expansion τ
$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e : \tau$	$\hat{e}$ has expansion e of type τ
$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \leadsto r : \tau \Mapsto \tau'$	$\hat{r}$ has expansion r taking values of type τ to values of type τ'
$\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \leadsto p : \tau \dashv\!\mid \hat{\Gamma}$	$\hat{p}$ has expansion p matching against τ generating hypotheses $\hat{\Gamma}$

Type Expansion

The *type expansion judgement*, $\hat{\Delta} \vdash \hat{\tau} \leadsto \tau\ \mathsf{type}$, is inductively defined by Rules (B.5) as before.

Typed Expression, Rule and Pattern Expansion

The *typed expression expansion* judgement, $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e : \tau$, and the *typed rule expansion judgement*, $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \leadsto r : \tau \Mapsto \tau'$ are defined mutually inductively by Rules (B.6) and Rule (B.7). The *typed pattern expansion judgement*, $\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \leadsto p : \tau \dashv\!\mid \hat{\Gamma}$, is inductively defined by Rules (B.8).

Rules (B.6a) through (B.6o) are adapted directly from $\mathsf{miniVerse}_{\mathbf{SE}}$, differing only in that the spTSM context, $\hat{\Phi}$, passes opaquely through them.

There is one new common unexpanded expression form in $\mathsf{miniVerse}_{\mathbf{S}}$: the unexpanded match form. Rule (B.6p) governs this form:

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Phi};\hat{\Psi}} \hat{e} \leadsto e : \tau \qquad \{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \leadsto r_i : \tau \Mapsto \tau'\}_{1 \le i \le n}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{match}\ \hat{e}\ \{\hat{r}_i\}_{1 \le i \le n} \leadsto \mathtt{match}[n](e; \{r_i\}_{1 \le i \le n}) : \tau'} \tag{B.6p}$$

The typed rule expansion judgement is defined by Rule (B.7), below:

$$\frac{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\!| \langle \mathcal{G}'; \Gamma' \rangle \qquad \hat{\Delta}\ \langle \mathcal{G} \uplus \mathcal{G}'; \Gamma \cup \Gamma' \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e : \tau'}{\hat{\Delta}\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \mathtt{urule}(\hat{p}.\hat{e}) \rightsquigarrow \mathtt{rule}(p.e) : \tau \Rrightarrow \tau'} \tag{B.7}$$

Because unexpanded terms mention only expression identifiers, which are given meaning by expansion to variables, the pattern typing rules must generate both an identifier expansion context, $\mathcal{G}'$, and a typing context, Γ'. In the second premise of the rule above, we update the "incoming" identifier expansion context, $\mathcal{G}$, with the new identifier expansions, $\mathcal{G}'$, and correspondingly, extend the "incoming" typing context, Γ, with the new typing hypotheses, Γ'.

Rules (B.8a) through (B.8e), reproduced below, define typed expansion of unexpanded patterns of common form.

$$\frac{}{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{x} \rightsquigarrow x : \tau \dashv\!\!| \langle \hat{x} \rightsquigarrow x; x : \tau \rangle} \tag{B.8a}$$

$$\frac{}{\hat{\Delta} \vdash_{\hat{\Phi}} \_ \rightsquigarrow \mathtt{wildp} : \tau \dashv\!\!| \langle \emptyset; \emptyset \rangle} \tag{B.8b}$$

$$\frac{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : [\mathtt{rec}(t.\tau)/t]\tau \dashv\!\!| \hat{\Gamma}}{\hat{\Delta} \vdash_{\hat{\Phi}} \mathtt{fold}(\hat{p}) \rightsquigarrow \mathtt{foldp}(p) : \mathtt{rec}(t.\tau) \dashv\!\!| \hat{\Gamma}} \tag{B.8c}$$

$$\frac{\begin{array}{c}\tau = \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \\ \{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p}_i \rightsquigarrow p_i : \tau_i \dashv\!\!| \hat{\Gamma}_i\}_{i \in L}\end{array}}{\hat{\Delta} \vdash_{\hat{\Phi}} \langle \{i \hookrightarrow \hat{p}_i\}_{i \in L} \rangle \rightsquigarrow \mathtt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \tau \dashv\!\!| \uplus_{i \in L} \hat{\Gamma}_i} \tag{B.8d}$$

$$\frac{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\!| \hat{\Gamma}}{\hat{\Delta} \vdash_{\hat{\Phi}} \mathtt{inj}[\ell](\hat{p}) \rightsquigarrow \mathtt{injp}[\ell](p) : \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\!\!| \hat{\Gamma}} \tag{B.8e}$$

Again, the unexpanded and expanded pattern forms in the conclusion correspond and the premises correspond to those of the corresponding pattern typing rule, i.e. Rules (B.4a) through (B.4e), respectively. The spTSM context, $\hat{\Phi}$, passes through these rules opaquely. In Rule (B.8d), the conclusion of the rule collects all of the identifier expansions and hypotheses generated by the subpatterns. We define $\hat{\Gamma}_i$ as shorthand for $\langle \mathcal{G}_i; \Gamma_i \rangle$ and $\uplus_{i \in L} \hat{\Gamma}_i$ as shorthand for

$$\langle \uplus_{i \in L} \mathcal{G}_i; \cup_{i \in L} \Gamma_i \rangle$$

By the definition of iterated extension of finite functions, we implicitly have that no identifiers or variables can be duplicated, i.e. that

$$\{\{\mathrm{dom}(\mathcal{G}_i) \cap \mathrm{dom}(\mathcal{G}_j) = \emptyset\}_{j \in L \setminus i}\}_{i \in L}$$

and

$$\{\{\mathrm{dom}(\Gamma_i) \cap \mathrm{dom}(\Gamma_j) = \emptyset\}_{j \in L \setminus i}\}_{i \in L}$$

spTSM Definition and Application Two rules remain: Rules (B.6q) and (B.8f), which define spTSM definition and application, respectively. These rules are defined in the next two subsections, respectively.

4.2.6 spTSM Definition

The spTSM definition form is

$$\texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for patterns by static}\ e_{\text{parse}}\ \texttt{in}\ \hat{e}$$

An unexpanded expression of this form defines a spTSM identified as $\hat{a}$ with *unexpanded type annotation* $\hat{\tau}$ and *parse function* e_{parse} for use within $\hat{e}$.

Rule (B.6q) defines typed expansion of spTSM definitions:

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \emptyset\,\emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResultSP)} \qquad e_{\text{parse}} \Downarrow e'_{\text{parse}} \qquad \hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi},\hat{a}\rightsquigarrow a \hookrightarrow \texttt{sptsm}(\tau;e'_{\text{parse}})} \hat{e} \rightsquigarrow e : \tau'}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for patterns by static}\ e_{\text{parse}}\ \texttt{in}\ \hat{e} \rightsquigarrow e : \tau'} \quad \text{(B.6q)}$$

This rule is similar to Rule (B.6n), which governs seTSM definitions. 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 the parse function, e_{parse}, is a closed expanded function of the following type:

   $$\texttt{parr(Body;ParseResultSE)}$$

   The assumed type `Body` is characterized as before by Condition B.16.

   `ParseResultSP`, like `ParseResultSE`, abbreviates a labeled sum type that distinguishes parse errors from successful parses:

   $$L_{\text{SP}} \overset{\text{def}}{=} \texttt{ParseError}, \texttt{SuccessP}$$

   $$\texttt{ParseResultSP} \overset{\text{def}}{=} \texttt{sum}[L_{\text{SP}}](\texttt{ParseError} \hookrightarrow \langle\rangle, \texttt{SuccessP} \hookrightarrow \texttt{PrPat})$$

   The type abbreviated `PrPat` classifies encodings of *proto-patterns*, $\grave{p}$. The syntax of proto-patterns, defined in Figure 4.4, will be described when we describe proto-expansion validation in Sec. 4.2.8. The mapping from proto-patterns to values of type `PrPat` is defined by the *proto-pattern encoding judgement*, $\grave{p} \downarrow_{\mathsf{PrPat}} e$. An inverse mapping is defined by the *proto-pattern decoding judgement*, $e \uparrow_{\mathsf{PrPat}} \grave{p}$.

Judgement Form	**Description**
$\grave{p} \downarrow_{\mathsf{PrPat}} e$	$\grave{p}$ has encoding e
$e \uparrow_{\mathsf{PrPat}} \grave{p}$	e has decoding $\grave{p}$

   Again, rather than picking a particular definition of `PrPat` and defining the judgements above inductively against it, we only state the following condition, which establishes an isomorphism between values of type `PrPat` and proto-patterns.

Condition B.23 (Proto-Pattern Isomorphism)**.**

(a) *For every $\grave{p}$, we have $\grave{p} \downarrow_{\mathsf{PrPat}} e_{proto}$ for some e_{proto} such that $\vdash e_{proto} : \texttt{PrPat}$ and e_{proto} val.*

(b) *If $\vdash e_{proto} : \texttt{PrPat}$ and e_{proto} val then $e_{proto} \uparrow_{\mathsf{PrPat}} \grave{p}$ for some $\grave{p}$.*

(c) *If $\grave{p} \downarrow_{\mathsf{PrPat}} e_{proto}$ then $e_{proto} \uparrow_{\mathsf{PrPat}} \grave{p}$.*

(d) *If $\vdash e_{proto} : \texttt{PrPat}$ and e_{proto} val and $e_{proto} \uparrow_{\mathsf{PrPat}} \grave{p}$ then $\grave{p} \downarrow_{\mathsf{PrPat}} e_{proto}$.*

(e) *If $\grave{p} \downarrow_{\mathsf{PrPat}} e_{proto}$ and $\grave{p} \downarrow_{\mathsf{PrPat}} e'_{proto}$ then $e_{proto} = e'_{proto}$.*

(f) *If $\vdash e_{proto} : \texttt{PrPat}$ and e_{proto} val and $e_{proto} \uparrow_{\mathsf{PrPat}} \grave{p}$ and $e_{proto} \uparrow_{\mathsf{PrPat}} \grave{p}'$ then $\grave{p} = \grave{p}'$.*

3. The third premise of Rule (B.6q) evaluates the parse function to a value.

4. The final premise of Rule (B.6q) extends the spTSM context, $\hat{\Phi}$, with the newly determined spTSM definition, and proceeds to assign a type, τ', and expansion, e, to $\hat{e}$. The conclusion of Rule (B.6q) assigns this type and expansion to the spTSM definition as a whole.

   spTSM contexts, $\hat{\Phi}$, are of the form $\langle \mathcal{A}; \Phi \rangle$, where $\mathcal{A}$ is a TSM identifier expansion context, defined previously, and Φ is a *spTSM definition context*.

   An *spTSM definition context*, Φ, is a finite function mapping each TSM name $a \in \text{dom}(\Phi)$ to an *expanded spTSM definition*, $a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}})$, where τ is the spTSM's type annotation, and e_{parse} is its parse function. We write $\Phi, a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}})$ when $a \notin \text{dom}(\Phi)$ for the extension of Φ that maps a to $a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}})$. We define $\hat{\Phi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}})$, when $\hat{\Phi} = \langle \mathcal{A}; \Phi \rangle$, as an abbreviation of

$$\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}}) \rangle$$

4.2.7 spTSM Application

The unexpanded pattern form for applying an spTSM named $\hat{a}$ to a literal form with literal body b is:

$$\hat{a}\ \text{`}b\text{`}$$

This stylized form is identical to the stylized form for seTSM application, differing in that appears within the syntax of unexpanded patterns, $\hat{p}$, rather than unexpanded expressions, $\hat{e}$.

Rule (B.8f), below, governs spTSM application.

$$\frac{\begin{array}{c} b \downarrow_{\mathsf{Body}} e_{\text{body}} \qquad \begin{array}{c}\hat{\Phi} = \hat{\Phi}', \hat{a} \rightsquigarrow a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}}) \\ e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}[\texttt{SuccessP}](e_{\text{proto}})\end{array} \qquad e_{\text{proto}} \uparrow_{\mathsf{PrPat}} \grave{p} \\ \mathsf{seg}(\grave{p})\ \mathsf{segments}\ b \qquad \grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\hat{\Delta}; \hat{\Phi}; b} \hat{\Gamma} \end{array}}{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{a}\ \text{`}b\text{`} \rightsquigarrow p : \tau \dashv\!\mid \hat{\Gamma}} \quad \text{(B.8f)}$$

This rule is similar to Rule (B.6o), which governs seTSM application. Its premises 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}.
3. The third premise applies the parse function e_{parse} to the encoding of the literal body. If parsing succeeds, then e_{proto} will be a value of type `PrPat` (assuming a well-formed spTSM context, by application of the Preservation assumption, Assumption B.9.) We call e_{proto} the *encoding of the proto-expansion*.

   If the parse function produces a value labeled `ParseError`, then typed expansion fails. No rule is necessary to handle this case.
4. The fourth premise decodes the encoding of the proto-expansion to produce the *proto-expansion*, $\grave{p}$, itself.
5. The fifth premise ensures that the proto-expansion induces a valid segmentation of b, i.e. that the spliced pattern locations are within bounds and non-overlapping.
6. The final premise of Rule (B.60) *validates* the proto-expansion and simultaneously generates the *final expansion*, e, and generates hypotheses $\hat{\Gamma}$, which appear in the conclusion of the rule. The proto-pattern validation judgement is discussed next.

4.2.8 Syntax of Proto-Expansions

Sort			**Operational Form**	**Stylized Form**	**Description**
PrTyp	$\grave{\tau}$	$::=$	$\cdots$	$\cdots$	(see Figure 3.6)
PrExp	$\grave{e}$	$::=$	$\cdots$	$\cdots$	(see Figure 3.6)
			$\texttt{prmatch}[n](\grave{e}; \{\grave{r}_i\}_{1\le i\le n})$	$\texttt{match}\ \grave{e}\ \{\grave{r}_i\}_{1\le i\le n}$	match
PrRule	$\grave{r}$	$::=$	$\texttt{prrule}(p.\grave{e})$	$p \Rightarrow \grave{e}$	rule
PrPat	$\grave{p}$	$::=$	$\texttt{prwildp}$	$\_$	wildcard pattern
			$\texttt{prfoldp}(p)$	$\texttt{fold}(p)$	fold pattern
			$\texttt{prtplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i\in L})$	$\langle\{i \hookrightarrow \grave{p}_i\}_{i\in L}\rangle$	labeled tuple pattern
			$\texttt{prinjp}[\ell](\grave{p})$	$\texttt{inj}[\ell](\grave{p})$	injection pattern
			$\texttt{splicedp}[m; n; \grave{\tau}]$	$\texttt{splicedp}[m; n; \grave{\tau}]$	spliced pattern ref.

Figure 4.4: Syntax of miniVerse$_{\mathsf{S}}$ proto-expansions

Figure 4.4 defines the syntax of proto-types, $\grave{\tau}$, proto-expressions, $\grave{e}$, proto-rules, $\grave{r}$, and proto-patterns, $\grave{p}$. Proto-expansion terms are identified up to α-equivalence in the usual manner.

Each expanded form, with the exception of the variable pattern form, maps onto a proto-expansion form. We refer to these collectively as the *common proto-expansion forms*. The mapping is given explicitly in Appendix B.3.

The main proto-expansion form of interest here, highlighted in yellow, is the proto-pattern form for *references to spliced unexpanded patterns*.

4.2.9 Proto-Expansion Validation

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

Judgement Form	Description
$\Delta \vdash^{\mathbb{T}} \grave{\tau} \leadsto \tau$ type	$\grave{\tau}$ has well-formed expansion τ
$\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau$	$\grave{e}$ has expansion e of type τ
$\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r} \leadsto r : \tau \Mapsto \tau'$	$\grave{r}$ has expansion r taking values of type τ to values of type τ'
$\grave{p} \leadsto p : \tau \dashv\vert^{\mathbb{P}} \hat{\Gamma}$	$\grave{p}$ has expansion p matching against τ generating assumptions $\hat{\Gamma}$

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 $\hat{\Delta}; \hat{\Phi}; b$. As in $\mathsf{miniVerse}_{\mathbf{SE}}$, their purpose is to "remember", during proto-expansion validation, the contexts and the literal body from the TSM application site (cf. Rules (B.6o) and (B.8f)), because these are necessary to validate references to spliced terms. We write $\mathsf{ts}(\mathbb{E})$ for the type splicing scene constructed by dropping unnecessary contexts from $\mathbb{E}$:

$$\mathsf{ts}(\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; \hat{\Phi}; b) = \hat{\Delta}; b$$

Proto-Type Validation

The *proto-type validation judgement*, $\Delta \vdash^{\mathbb{T}} \grave{\tau} \leadsto \tau$ type, is inductively defined by Rules (B.9), which were already described in Sec. 3.2.10.

Proto-Expansion Expression and Rule Validation

The *proto-expression validation judgement*, $\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau$, and the *proto-rule validation judgement*, $\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r} \leadsto r : \tau \Mapsto \tau'$, are defined mutually inductively with Rules (B.6) and Rule (B.7) by Rules (B.10) and Rule (B.11), respectively.

Rules (B.10a) through (B.10n) were described in Sec. 3.2.10. Rule (B.10o) governs match proto-expressions:

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau \qquad \{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r}_i \leadsto r_i : \tau \Mapsto \tau'\}_{1 \le i \le n}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prmatch}[n](\grave{e}; \{\grave{r}_i\}_{1 \le i \le n}) \leadsto \mathtt{match}[n](e; \{r_i\}_{1 \le i \le n}) : \tau'} \tag{B.10o}$$

Rule (B.11) governs proto-rules:

$$\frac{\Delta \vdash p : \tau \dashv\vert\ \Gamma \qquad \Delta\ \Gamma \cup \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prrule}(p.\grave{e}) \leadsto \mathtt{rule}(p.e) : \tau \Mapsto \tau'} \tag{B.11}$$

Notice that proto-rules bind expanded patterns, rather than proto-patterns. This is because proto-rules appear in proto-expressions, which are generated by seTSMs. Proto-patterns are generated exclusively by spTSMs.

Proto-Pattern Validation

spTSMs generate candidate expansions of proto-pattern form, as described in Sec. 4.2.7. The *proto-pattern validation judgement*, $\grave{p} \rightsquigarrow p : \tau \dashv\!|^{\mathbb{P}} \hat{\Gamma}$, which appears as the final premise of Rule (B.8f), validates proto-patterns and simultaneously generates the final expansion, p, and the unexpanded typing hypotheses $\hat{\Gamma}$.

The proto-pattern validation judgement is defined mutually inductively with Rules (B.8) by Rules (B.12), reproduced below.

$$\frac{}{\texttt{prwildp} \rightsquigarrow \texttt{wildp} : \tau \dashv\!|^{\mathbb{P}} \langle\emptyset;\emptyset\rangle} \tag{B.12a}$$

$$\frac{\grave{p} \rightsquigarrow p : [\texttt{rec}(t.\tau)/t]\tau \dashv\!|^{\mathbb{P}} \hat{\Gamma}}{\texttt{prfoldp}(\grave{p}) \rightsquigarrow \texttt{foldp}(p) : \texttt{rec}(t.\tau) \dashv\!|^{\mathbb{P}} \hat{\Gamma}} \tag{B.12b}$$

$$\frac{\begin{array}{c}\tau = \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L}) \\ \{\grave{p}_i \rightsquigarrow p_i : \tau_i \dashv\!|^{\mathbb{P}} \hat{\Gamma}_i\}_{i\in L}\end{array}}{\texttt{prtplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i\in L}) \rightsquigarrow \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i\in L}) : \tau \dashv\!|^{\mathbb{P}} \uplus_{i\in L}\hat{\Gamma}_i} \tag{B.12c}$$

$$\frac{\grave{p} \rightsquigarrow p : \tau \dashv\!|^{\mathbb{P}} \hat{\Gamma}}{\texttt{prinjp}[\ell](\grave{p}) \rightsquigarrow \texttt{injp}[\ell](p) : \texttt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i\in L}; \ell \hookrightarrow \tau) \dashv\!|^{\mathbb{P}} \hat{\Gamma}} \tag{B.12d}$$

$$\frac{\emptyset \vdash^{\hat{\Delta};b} \grave{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \mathsf{parseUPat}(\mathsf{subseq}(b;m;n)) = \hat{p} \qquad \hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!| \hat{\Gamma}}{\texttt{splicedp}[m;n;\grave{\tau}] \rightsquigarrow p : \tau \dashv\!|^{\hat{\Delta};\hat{\Phi};b} \hat{\Gamma}} \tag{B.12e}$$

Rules (B.12a) through (B.12d) govern proto-patterns of common form, and behave like the corresponding pattern typing rules, i.e. Rules (B.4b) through (B.4e). Rule (B.12e) governs references to spliced unexpanded patterns. The first premise validates the type annotation. The second premise parses the indicated subsequence of the literal body, b, to produce the referenced unexpanded pattern, $\hat{p}$, and the third premise types and expands $\hat{p}$ under the spTSM context $\hat{\Phi}$ from the spTSM application site, generating the hypotheses $\hat{\Gamma}$. These are the hypotheses generated in the conclusion of the rule.

Hypotheses can be generated only by spliced subpatterns, so there is no proto-pattern form corresponding to variable patterns. This achieves the prohibition on hidden bindings described in Sec. 4.1.5. We consider this invariant formally below.

4.2.10 Metatheory

The following theorem establishes that typed pattern expansion produces an expanded pattern that matches values of the specified type and generates the specified hypotheses. We must mutually state the corresponding proposition about proto-patterns, because the relevant judgements are mutually defined.

Theorem B.26 (Typed Pattern Expansion)**.**

1. *If* $\langle\mathcal{D};\Delta\rangle \vdash_{\langle\mathcal{A};\Phi\rangle} \hat{p} \rightsquigarrow p : \tau \dashv\!| \langle\mathcal{G};\Gamma\rangle$ *then* $\Delta \vdash p : \tau \dashv\!| \Gamma$.
2. *If* $\grave{p} \rightsquigarrow p : \tau \dashv\!|^{\langle\mathcal{D};\Delta\rangle;\langle\mathcal{A};\Phi\rangle;b} \langle\mathcal{G};\Gamma\rangle$ *then* $\Delta \vdash p : \tau \dashv\!| \Gamma$.

Proof. By mutual rule induction on Rules (B.8) and Rules (B.12). The full proof is given in Appendix B.4.2. We will reproduce only the interesting cases below.

1. The only interesting case in the proof of part 1 is the case for spTSM application. In the following, let $\hat{\Delta} = \langle \mathcal{D}; \Delta \rangle$ and $\hat{\Gamma} = \langle \mathcal{G}; \Gamma \rangle$ and $\hat{\Phi} = \langle \mathcal{A}; \Phi \rangle$.

   Case (B.8f).

(1) $\hat{p} = \hat{a}$ ‘b‘	by assumption
(2) $\mathcal{A} = \mathcal{A}', \hat{a} \rightsquigarrow a$	by assumption
(3) $\Phi = \Phi', a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}})$	by assumption
(4) $b \downarrow_{\mathsf{Body}} e_{\text{body}}$	by assumption
(5) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}[\texttt{SuccessP}](e_{\text{proto}})$	by assumption
(6) $e_{\text{proto}} \uparrow_{\mathsf{PrPat}} \grave{p}$	by assumption
(7) $\grave{p} \rightsquigarrow p : \tau \dashv\vert^{\hat{\Delta}; \langle \mathcal{A}; \Phi \rangle; b} \langle \mathcal{G}; \Gamma \rangle$	by assumption
(8) $\Delta \vdash p : \tau \dashv\vert \Gamma$	by IH, part 2 on (7)

2. The only interesting case in the proof of part 2 is the case for spliced patterns. In the following, let $\hat{\Gamma} = \langle \mathcal{G}; \Gamma \rangle$ and $\hat{\Delta} = \langle \mathcal{D}; \Delta \rangle$ and $\hat{\Phi} = \langle \mathcal{A}; \Phi \rangle$.

   Case (B.12e).

(1) $\grave{p} = \texttt{splicedp}[m; n; \grave{\tau}]$	by assumption
(2) $\emptyset \vdash^{\hat{\Delta}; b} \grave{\tau} \rightsquigarrow \tau\ \mathsf{type}$	by assumption
(3) $\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{p}$	by assumption
(4) $\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\vert \hat{\Gamma}$	by assumption
(5) $\Delta \vdash p : \tau \dashv\vert \Gamma$	by IH, part 1 on (4)

The mutual induction can be shown to be well-founded by showing that the following numeric metric on the judgements that we induct on is decreasing:

$$\|\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\vert \hat{\Gamma}\| = \|\hat{p}\|$$
$$\|\grave{p} \rightsquigarrow p : \tau \dashv\vert^{\hat{\Delta}; \hat{\Phi}; b} \hat{\Gamma}\| = \|b\|$$

where $\|b\|$ is the length of b and $\|\hat{p}\|$ is the sum of the lengths of the literal bodies in $\hat{p}$ (see Appendix B.2.1.)

The only case in the proof of part 1 that invokes part 2 is Case (B.8f). There, we have that the metric remains stable:

$$\begin{aligned} &\|\hat{\Delta} \vdash_{\hat{\Phi}} \hat{a}\ \text{‘}b\text{‘} \rightsquigarrow p : \tau \dashv\vert \hat{\Gamma}\| \\ =&\|\grave{p} \rightsquigarrow p : \tau \dashv\vert^{\hat{\Delta}; \hat{\Phi}; b} \hat{\Gamma}\| \\ =&\|b\| \end{aligned}$$

The only case in the proof of part 2 that invokes part 1 is Case (B.12e). There, we have that $\mathsf{parseUPat}(\mathsf{subseq}(b; m; n)) = \hat{p}$ and the IH is applied to the judgement $\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\vert \hat{\Gamma}$. Because the metric is stable when passing from part 1 to part 2, we must have that it is strictly decreasing in the other direction:

$$\|\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\vert \hat{\Gamma}\| < \|\texttt{splicedp}[m; n; \grave{\tau}] \rightsquigarrow p : \tau \dashv\vert^{\hat{\Delta}; \hat{\Phi}; b} \hat{\Gamma}\|$$

i.e. by the definitions above,

$$\|\hat{p}\| < \|b\|$$

This is established by appeal to Condition B.17, which states that subsequences of b are no longer than b, and the following condition, which states that an unexpanded pattern constructed by parsing a textual sequence b is strictly smaller, as measured by the metric defined above, than the length of b, because some characters must necessarily be used to apply the pattern TSM and delimit each literal body.

Condition B.13 (Pattern Parsing Monotonicity)**.** *If* $\mathsf{parseUPat}(b) = \hat{p}$ *then* $\|\hat{p}\| < \|b\|$.

Combining Conditions B.17 and B.13, we have that $\|\hat{e}\| < \|b\|$ as needed. □

Finally, the following theorem establishes that typed expression and rule expansion produces expanded expressions and rules of the same type under the same contexts. Again, it must be stated mutually with the corresponding theorem about candidate expansion expressions and rules because the judgements are mutually defined.

Theorem 4.9 (Typed Expansion)**.**

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

Proof. By mutual rule induction on Rules (B.6), Rule (B.7), Rules (B.10) and Rule (B.11). The full proof is given in Appendix B.4. We will reproduce only the cases that have to do with pattern matching below.

1. In the following cases, let $\hat{\Delta} = \langle\mathcal{D};\Delta\rangle$ and $\hat{\Gamma} = \langle\mathcal{G};\Gamma\rangle$.
   (a) The only cases in the proof of part 1(a) that have to do with pattern matching are the cases involving the unexpanded match expression and spTSM definition.

   Case (B.6p)**.**

(1) $\hat{e} = \mathtt{match}\ \hat{e}'\ \{\hat{r}_i\}_{1\leq i\leq n}$	by assumption
(2) $e = \mathtt{match}[n](e';\{r_i\}_{1\leq i\leq n})$	by assumption
(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \rightsquigarrow e' : \tau'$	by assumption
(4) $\{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \rightsquigarrow r_i : \tau' \Mapsto \tau\}_{1\leq i\leq n}$	by assumption
(5) $\Delta\ \Gamma \vdash e' : \tau'$	by IH, part 1(a) on (3)
(6) $\{\Delta\ \Gamma \vdash r_i : \tau' \Mapsto \tau\}_{1\leq i\leq n}$	by IH, part 1(b) over (4)
(7) $\Delta\ \Gamma \vdash \mathtt{match}[n](\tau;e')\{r_i\}_{1\leq i\leq n} : \tau$	by Rule (B.21) on (5) and (6)

   Case (B.6q)**.**

   (1) $\hat{e} = \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}'\ \mathtt{for\ patterns\ by\ static}\ e_{\text{parse}}\ \mathtt{in}\ \hat{e}'$
   by assumption

(2) $\hat{\Delta} \vdash \hat{\tau}' \leadsto \tau'$ type — by assumption

(3) $\emptyset\,\emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResultSP)}$ — by assumption

(4) $\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi},\hat{a}\leadsto a \hookrightarrow \texttt{sptsm}(\tau';e_{\text{parse}})} \hat{e}' \leadsto e : \tau$ — by assumption

(5) $\Delta \vdash \tau'$ type — by Lemma B.24 to (2)

(6) $\Delta\,\Gamma \vdash e : \tau$ — by IH, part 1(a) on (4)

(b) There is only one case.

Case (B.7).

(1) $\hat{r} = \hat{p} \Rightarrow \hat{e}$ — by assumption

(2) $r = \texttt{rule}(p.e)$ — by assumption

(3) $\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \leadsto p : \tau \dashv\!\mid \langle \mathcal{A}';\Gamma' \rangle$ — by assumption

(4) $\hat{\Delta}\,\langle \mathcal{A} \uplus \mathcal{A}';\Gamma \cup \Gamma' \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e : \tau'$ — by assumption

(5) $\Delta \vdash p : \tau \dashv\!\mid \Gamma'$ — by Theorem B.26, part 1 on (3)

(6) $\Delta\,\Gamma \cup \Gamma' \vdash e : \tau'$ — by IH, part 1(a) on (4)

(7) $\Delta\,\Gamma \vdash \texttt{rule}(p.e) : \tau \Mapsto \tau'$ — by Rule (B.3) on (5) and (6)

2. In the following, let $\hat{\Delta} = \langle \mathcal{D};\Delta_{\text{app}} \rangle$ and $\hat{\Gamma} = \langle \mathcal{G};\Gamma_{\text{app}} \rangle$.

(a) The only case in the proof of part 2(a) that has to do with pattern matching is the case involving the match proto-expression.

Case (B.10o).

(1) $\grave{e} = \texttt{prmatch}[n](\grave{e}';\{\grave{r}_i\}_{1\le i\le n})$ — by assumption

(2) $e = \texttt{match}[n](e';\{r_i\}_{1\le i\le n})$ — by assumption

(3) $\Delta\,\Gamma \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e}' \leadsto e' : \tau'$ — by assumption

(4) $\{\Delta\,\Gamma \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{r}_i \leadsto r_i : \tau' \Mapsto \tau\}_{1\le i\le n}$ — by assumption

(5) $\Delta \cap \Delta_{\text{app}} = \emptyset$ — by assumption

(6) $\text{dom}(\Gamma) \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset$ — by assumption

(7) $\Delta \cup \Delta_{\text{app}}\;\Gamma \cup \Gamma_{\text{app}} \vdash e' : \tau'$ — by IH, part 2(a) on (3), (5) and (6)

(8) $\Delta \cup \Delta_{\text{app}}\;\Gamma \cup \Gamma_{\text{app}} \vdash r : \tau' \Mapsto \tau$ — by IH, part 2(b) on (4), (5) and (6)

(9) $\Delta \cup \Delta_{\text{app}}\;\Gamma \cup \Gamma_{\text{app}} \vdash \texttt{match}[n](e';\{r_i\}_{1\le i\le n}) : \tau$ — by Rule (B.21) on (7) and (8)

(b) There is only one case.

Case (B.11).

(1) $\grave{r} = \texttt{prrule}(p.\grave{e})$ — by assumption

(2) $r = \texttt{rule}(p.e)$ — by assumption

(3) $\Delta \vdash p : \tau \dashv\!\mid \Gamma$ — by assumption

(4) $\Delta\,\Gamma \cup \Gamma \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \leadsto e : \tau'$ — by assumption

(5) $\Delta \cap \Delta_{\text{app}} = \emptyset$ — by assumption

(6) $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma) = \emptyset$ — by identification convention

(7) $\mathsf{dom}(\Gamma_{\mathsf{app}}) \cap \mathsf{dom}(\Gamma) = \emptyset$ — by identification convention

(8) $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma_{\mathsf{app}}) = \emptyset$ — by assumption

(9) $\mathsf{dom}(\Gamma \cup \Gamma) \cap \mathsf{dom}(\Gamma_{\mathsf{app}}) = \emptyset$ — by standard finite set definitions and identities on (6), (7) and (8)

(10) $\Delta \cup \Delta_{\mathsf{app}}\ \Gamma \cup \Gamma \cup \Gamma_{\mathsf{app}} \vdash e : \tau'$ — by IH, part 2(a) on (4), (5) and (9)

(11) $\Delta \cup \Delta_{\mathsf{app}}\ \Gamma \cup \Gamma_{\mathsf{app}} \cup \Gamma \vdash e : \tau'$ — by exchange of Γ and Γ_{app} on (10)

(12) $\Delta \cup \Delta_{\mathsf{app}}\ \Gamma \cup \Gamma_{\mathsf{app}} \vdash \mathtt{rule}(p.e) : \tau \Mapsto \tau'$ — by Rule (B.3) on (3) and (11)

The mutual induction can be shown to be well-founded essentially as described in Sec. 3.2.11. Appendix B.4 gives the complete details. □

Abstract Reasoning Principles

The following theorem summarizes the abstract reasoning principles available to programmers when applying an spTSM. Descriptions of labeled clauses are given inline.

Theorem B.33 (spTSM Abstract Reasoning Principles). *If $\hat{\Delta} \vdash_{\hat{\Phi}} \hat{a}$ ‘b‘ $\rightsquigarrow p : \tau \dashv\!\!\mid \hat{\Gamma}$ 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**) $\hat{\Phi} = \hat{\Phi}', \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{sptsm}(\tau; e_{parse})$ and $\Delta \vdash p : \tau \dashv\!\!\mid \Gamma$*

   The final expansion matches values of the type specified by the spTSM's type annotation.

2. $b \downarrow_{\mathsf{Body}} e_{body}$
3. $e_{parse}(e_{body}) \Downarrow \mathtt{inj}[\mathsf{SuccessP}](e_{proto})$
4. $e_{proto} \uparrow_{\mathsf{PrPat}} \grave{p}$
5. *(**Segmentation**)* $\mathsf{seg}(\grave{p})$ segments b

   The segmentation determined by the proto-expansion actually segments the literal body (i.e. each segment is in-bounds and the segments are non-overlapping.)

6. $\mathsf{summary}(\grave{p}) = \{\mathtt{splicedt}[n'_i; m'_i]\}_{0 \leq i < n_{ty}} \cup \{\mathtt{splicedp}[m_i; n_i; \grave{\tau}_i]\}_{0 \leq i < n_{pat}}$
7. *(**Typing 2**)* $\{\hat{\Delta} \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow \tau'_i\ \mathsf{type}\}_{0 \leq i < n_{ty}}$ *and* $\{\Delta \vdash \tau'_i\ \mathsf{type}\}_{0 \leq i < n_{ty}}$

   Each spliced type has a well-formed expansion at the application site.

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

   Each type annotation on a reference to a spliced pattern has a well-formed expansion at the application site.

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

Each spliced pattern has a well-typed expansion that matches values of the type indicated by the corresponding type annotation in the splice summary.

10. *(**No Hidden Bindings**)* $\mathcal{G} = \biguplus_{0 \leq i < n_{pat}} \mathcal{G}_i$ *and* $\Gamma = \bigcup_{0 \leq i < n_{pat}} \Gamma_i$

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

Proof. The proof relies on a lemma about decomposing proto-patterns. The proof is given in Appendix B.4.4. □

Chapter 5

Parametric TSMs (pTSMs)

You know me, I gotta put in a big tree.

— Bob Ross, *The Joy of Painting*

This chapter introduces *parametric TSMs* (pTSMs). Parametric TSMs can be defined over a parameterized family of types, rather than just a single type, and the expansions that they generate can refer to supplied type and module parameters.

This chapter is organized like the preceding chapters. We begin in Sec. 5.1 by introducing parametric TSMs by example in VerseML. In particular, we discuss type parameters in Sec. 5.1.1 and module parameters in Sec. 5.1.2. We then develop a reduced calculus of parametric TSMs, $\mathsf{miniVerse}_{\mathbf{P}}$, in Sec. 5.2.

5.1 Parametric TSMs By Example

5.1.1 Type Parameters

Recall from Sec. 2.3.1 the definition of the type-parameterized family of list types:

```
type list('a) = rec(self => Nil + Cons of 'a * self)
```

Figure 5.1 defines a *parametric expression TSM* (peTSM) and a *parametric pattern TSM* (ppTSM), both named `$list`. These TSMs operate uniformly over this family of types.

```
1 syntax $list('a) at list('a) for expressions by
2   static fn(b : body) : parse_result(proto_expr) => (* ... *)
3 and for patterns by
4   static fn(b : body) : parse_result(proto_pat) => (* ... *)
5 end
```

Figure 5.1: The type-parameterized `$list` TSMs.

Line 1 specifies a single type parameter, `'a`. This type parameter appears in the type annotation, which establishes that:

1. The peTSM `$list`, when applied to a type `T` and a generalized literal form, can only generate expansions of type `list(T)`.
2. The ppTSM `$list`, when applied to a type `T` and a generalized literal form, can only generate expansions that match values of type `list(T)`.

For example, we can apply `$list` to `int` and a generalized literal form delimited by square brackets as follows:

```
val x = $list int [x, y :: xs]
```

The parse function (elided above for concision) segments the literal body into spliced expressions. The trailing spliced expression is prefixed by two colons (`::`), which the TSM takes to mean that it should be the tail of the list. The final expansion of the example above is equivalent to the following when the list value constructors are in scope:

```
val x = Cons(x, Cons(y, xs))
```

As in the preceding chapters, the expansion itself must use the explicit **`fold`** and **`inj`** operators rather than the list value constructors `Cons` and `Nil` due to the prohibition on context dependence.

5.1.2 Module Parameters

We can finally address the inconvenience of needing to use explicit **`fold`** and **`inj`** operators by defining a module-parameterized TSM.

Recall that in Figure 2.1, we defined a signature `LIST` that exported the definition of `list` and specified the list value constructors (and some other values.) The definition of `$list'` shown in Figure 5.2 takes modules matching this signature as an additional parameter.

```
syntax $list' (L : LIST) 'a at 'a L.list for expressions by
  static fn(b : body) : parse_result(proto_expr) => (* ... *)
for patterns by
  static fn(b : body) : parse_result(proto_pat) => (* ... *)
end
```

Figure 5.2: The type- and module-parameterized `$list'` TSMs

We can apply `$list'` to the module `List` and the type `int` as follows:

```
val y = $list' List int [3, 4, 5]
val x = $list' List int [1, 2 :: y]
```

The expansion is:

```
val y = List.Cons(3, List.Cons(4, List.Cons(5, List.Nil)))
val x = List.Cons(1, List.Cons(2, y))
```

There is no need to use explicit **`fold`** and **`inj`** operators in this expansion, because the expansion projects the constructors out of the provided module parameter. The TSM itself did not assume that the module would be named `List` (internally, the proto-expansion refers to it as `L`.)

This makes matters simpler for the TSM provider, but there is a syntactic cost associated with supplying a module parameter at each TSM application site. To reduce this cost, VerseML supports partial parameter application in TSM abbreviations. For example, we can define `$list` by partially applying `$list'` as follows:

```
let syntax $list = $list' List
```

(This abbreviates both the expression and pattern TSMs – sort qualifiers can be added to restrict the abbreviation if desired.)

Module parameters also allow us to define TSMs that operate uniformly over module-parameterized families of abstract types. For example, the module-parameterized TSM `$r` defined in Figure 5.3 supports the POSIX regex syntax for any type `R.t` where `R : RX`.

```
syntax $r(R : RX) at R.t by
  static fn(b : body) : parse_result(proto_expr) => (* ... *)
end
```

Figure 5.3: The module-parameterized TSM `$r`

For example, given `R1 : RX`, we can apply `$r` as follows:

```
let dna = $r R1 /A|T|G|C/
```

The final expansion of this term is:

```
let dna = R1.Or(R1.Str "A", R1.Or(R1.Str "T",
          R1.Or(R1.Str "G", R1.Str "C")))
```

To be clear: parameters are available to the generated expansion, but they are not available to the parse function that generates the expansion. For example, the following TSM definition is not well-typed because it refers to `M` from within the parse function:

```
syntax $badM(M : A) at T by
  static fn(b : body) => let x = M.x in (* ... *)
end
```

(In the next chapter, we will define a mechanism that gives parse functions access to a common static environment.)

5.2 miniVerse$_{\mathbf{P}}$

We will now define a reduced dialect of VerseML called miniVerse$_{\mathbf{P}}$ that supports parametric expression and pattern TSMs (peTSMs and ppTSMs.) This language, like miniVerse$_{\mathbf{S}}$, consists of an unexpanded language (UL) defined by typed expansion to an expanded language (XL). The full definition of miniVerse$_{\mathbf{P}}$ is given in Appendix C – we will detail only particularly interesting constructs below.

5.2.1 Syntax of the Expanded Language (XL)

Figure 5.4 defines the syntax of the *expanded module language*. Figure 5.5 defines the syntax of the *expanded type construction language*. Figure 5.6 defines the syntax of the *expanded expression language*.

Sort			Operational Form	Description
Sig	σ	$::=$	$\texttt{sig}\{\kappa\}(u.\tau)$	signature
Mod	M	$::=$	X	module variable
			$\texttt{struct}(c;e)$	structure
			$\texttt{seal}\{\sigma\}(M)$	seal
			$\texttt{mlet}\{\sigma\}(M;X.M)$	definition

Figure 5.4: Syntax of signatures and module expressions in miniVerse$_{\mathbf{P}}$

Sort			Operational Form	Description
Kind	κ	$::=$	k	kind variable
			$\texttt{darr}(\kappa;u.\kappa)$	dependent function
			$\texttt{unit}$	nullary product
			$\texttt{dprod}(\kappa;u.\kappa)$	dependent product
			$\texttt{Type}$	type
			$\texttt{S}(\tau)$	singleton
Con	c,τ	$::=$	u	construction variable
			t	type variable
			$\texttt{abs}(u.c)$	abstraction
			$\texttt{app}(c;c)$	application
			$\texttt{triv}$	trivial
			$\texttt{pair}(c;c)$	pair
			$\texttt{prl}(c)$	left projection
			$\texttt{prr}(c)$	right projection
			$\texttt{parr}(\tau;\tau)$	partial function
			$\texttt{all}\{\kappa\}(u.\tau)$	polymorphic
			$\texttt{rec}(t.\tau)$	recursive
			$\texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})$	labeled product
			$\texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i\in L})$	labeled sum
			$\texttt{con}(M)$	construction component

Figure 5.5: Syntax of kinds and constructions in miniVerse$_{\mathbf{P}}$. By convention, we choose the metavariable τ for constructions that, in well-formed terms, must necessarily be of kind $\texttt{T}$, and the metavariable c otherwise. Similarly, we use construction variables t to stand for constructions of kind $\texttt{T}$, and construction variables u otherwise. Kind variables, k, are necessary only for the metatheory.

Sort			Operational Form	Description
Exp	e	$::=$	x	variable
			$\texttt{lam}\{\tau\}(x.e)$	abstraction
			$\texttt{ap}(e;e)$	application
			$\texttt{clam}\{\kappa\}(u.e)$	construction abstraction
			$\texttt{cap}\{c\}(e)$	construction application
			$\texttt{fold}(e)$	fold
			$\texttt{unfold}(e)$	unfold
			$\texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L})$	labeled tuple
			$\texttt{prj}[\ell](e)$	projection
			$\texttt{inj}[\ell](e)$	injection
			$\texttt{match}[n](e;\{r_i\}_{1 \le i \le n})$	match
			$\texttt{val}(M)$	value component
Rule	r	$::=$	$\texttt{rule}(p.e)$	rule
Pat	p	$::=$	x	variable pattern
			$\texttt{wildp}$	wildcard pattern
			$\texttt{foldp}(p)$	fold pattern
			$\texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L})$	labeled tuple pattern
			$\texttt{injp}[\ell](p)$	injection pattern

Figure 5.6: Syntax of expanded expressions, rules and patterns in miniVerse$_\mathbf{P}$

5.2.2 Statics of the Expanded Language

The module and type construction languages are based closely on those defined by Harper in *PFPL* [61]. These languages, in turn, are based on the languages developed by Lee et al. [78], and also by Dreyer [37]. All of these incorporate Stone and Harper's *dependent singleton kinds* formalism to track type identity [117]. The expression language is similar to that of miniVerse$_\mathbf{S}$, defined in Chapter 4.

The *statics of the expanded language* is defined by a collection of judgements that we organize into three groups.

The first group of judgements, which we refer to as the *statics of the expanded module language*, define the statics of expanded signatures and module expressions.

Judgement Form	Description
$\Omega \vdash \sigma\ \mathsf{sig}$	σ is a signature
$\Omega \vdash \sigma \equiv \sigma'$	σ and σ' are definitionally equal signatures
$\Omega \vdash \sigma <: \sigma'$	σ is a sub-signature of σ'
$\Omega \vdash M : \sigma$	M matches σ
$\Omega \vdash M\ \mathsf{mval}$	M is, or stands for, a module value

The second group of judgements, which we refer to as the *statics of the expanded type construction language*, define the statics of expanded kinds and constructions.

Judgement Form	Description
$\Omega \vdash \kappa$ kind	κ is a kind
$\Omega \vdash \kappa \equiv \kappa'$	κ and κ' are definitionally equal kinds
$\Omega \vdash \kappa <:: \kappa'$	κ is a subkind of κ'
$\Omega \vdash c :: \kappa$	c has kind κ
$\Omega \vdash c \equiv c' :: \kappa$	c and c' are equivalent as constructions of kind κ

The third group of judgements, which we refer to as the *statics of the expanded expression language*, define the statics of types, expanded expressions, rules and patterns. Types are constructions of kind `Type`. We use the metavariable τ rather than c for types.

Judgement Form	Description
$\Omega \vdash \tau <: \tau'$	τ is a subtype of τ'
$\Omega \vdash e : \tau$	e is assigned type τ
$\Omega \vdash r : \tau \mapsto \tau'$	r takes values of type τ to values of type τ'
$\Omega \vdash p : \tau \dashv\!\!\mid \Omega'$	p matches values of type τ and generates hypotheses Ω'

A *unified context*, Ω, is a finite function over module, expression and construction variables. We write

- $\Omega, X : \sigma$ when $X \notin \text{dom}(\Omega)$ and $\Omega \vdash \sigma$ sig for the extension of Ω with a mapping from X to the hypothesis $X : \sigma$.
- $\Omega, x : \tau$ when $x \notin \text{dom}(\Omega)$ and $\Omega \vdash \tau ::$ `Type` for the extension of Ω with a mapping from x to the hypothesis $x : \tau$
- $\Omega, u :: \kappa$ when $u \notin \text{dom}(\Omega)$ and $\Omega \vdash \kappa$ kind for the extension of Ω with a mapping from u to the hypothesis $u :: \kappa$

A well-formed unified context is one that can be constructed by some sequence of such extensions, starting from the empty context, $\emptyset$. We identify unified contexts up to exchange and contraction in the usual manner.

The complete set of rules is given in Appendix C.1.2. A comprehensive introductory account of these constructs is beyond the scope of this work (see [61].) Instead, let us summarize the key features of the expanded language by example.

Modules take the form `struct(`c`;`e`)`, following a *phase-splitting* approach – the construction components of the module are "tupled" into a single construction component, c, and the value components of the module are "tupled" into a single value component, e [62]. Signatures, σ, are also split in this way – a single *kind*, κ, classifies the construction component and a single type, τ, classifies the value component of the classified module. The type can refer to the construction component through a mediating construction variable, u. The key rule is reproduced below:

$$\frac{\Omega \vdash c :: \kappa \qquad \Omega \vdash e : [c/u]\tau}{\Omega \vdash \texttt{struct}(c;e) : \texttt{sig}\{\kappa\}(u.\tau)} \tag{C.4c}$$

For example, consider the VerseML signature and the corresponding miniVerse$_\mathsf{P}$ signature in Figure 5.7. The kind on the right (Lines 1-3) is a *dependent product kind* and the type (Lines 4-5) is a product type. Let us consider these in turn.

```
1 sig
2   type t
3   type t' = t * t
4   val x : t
5   val y : t -> t'
6 end
```

$$
\begin{array}{l}
\texttt{sig\{dprod(} \\
\quad \texttt{Type;}\, t. \\
\quad \texttt{S(prod}[\texttt{1;2}]\texttt{(1} \hookrightarrow t\texttt{;2} \hookrightarrow t\texttt{)))} \\
\texttt{\}(}u.\texttt{prod}[\texttt{x;y}]\texttt{(x} \hookrightarrow \texttt{prl(}u\texttt{);} \\
\quad \texttt{y} \hookrightarrow \texttt{parr(prl(}u\texttt{);prr(}u\texttt{)))} \\
\texttt{)}
\end{array}
$$

Figure 5.7: A VerseML signature and the corresponding miniVerse$_{\mathbf{P}}$ signature

On Lines 2-3 (left), we specified an abstract type component `t`, and then a translucent type component `t'` equal to `t * t`. Abstract type components have kind `Type`, so the first component of the dependent product kind is `Type` (Line 2, right). The construction variable t stands for the first component in the second component of the dependent product kind. The second component is not held abstract, so it is classified by a corresponding *singleton kind*, rather than by the kind `Type` (Line 3, right). A singleton kind $\texttt{S}(\tau)$ classifies only those types definitionally equal to τ. A subkinding system is necessary to ensure that constructions of singleton kind can appear where a construction of kind `Type` is needed – the key rule is reproduced below:

$$\frac{\Omega \vdash \tau :: \texttt{Type}}{\Omega \vdash \texttt{S}(\tau) <:: \texttt{Type}} \tag{C.8e}$$

Lines 4-5 (right) define a product type that classifies the value component of matching modules. The construction variable u stands for the construction component of the matching module. The left- and right-projection operations $\texttt{prl}(c)$ and $\texttt{prr}(c)$ on the right correspond to `t` and `t'` on the left. (In practice, we would use labeled dependent product kinds, but for simplicity, we stick to binary dependent product kinds here.)

Consider another example: the VerseML `LIST` signature from Figure 2.1, partially reproduced below:

```
sig
  type list('a) = rec(self => Nil + Cons of 'a * self)
  val Nil : list('a)
  val Cons : 'a * list('a) -> list('a)
  (* ... *)
end
```

This signature corresponds to the miniVerse$_{\mathbf{P}}$ signature $\sigma_{\texttt{LIST}}$ defined in Figure 5.8.

Here, the signature specifies only a single construction component, so no tupling of the construction component is necessary. This single construction component is a type function, so it has dependent function kind: the argument kind is `Type` and the return kind is a singleton kind, because the type function is not abstract. (Had we held the type function abstract, its kind would instead be `darr(Type; _.Type)`.)

At the top level, a program consists of a module expression, M. The module let binding form allows the programmer to bind a module to a module variable, X:

$$\frac{\Omega \vdash M : \sigma \qquad \Omega \vdash \sigma' \;\mathsf{sig} \qquad \Omega, X : \sigma \vdash M' : \sigma'}{\Omega \vdash \texttt{mlet}\{\sigma'\}(M; X.M') : \sigma'} \tag{C.4e}$$

$$
\begin{aligned}
&\sigma_{\texttt{LIST}} \stackrel{\text{def}}{=} \texttt{sig}\{\kappa_{\texttt{LIST}}\}(\mathit{list}.\tau_{\texttt{LIST}})\\
&\kappa_{\texttt{LIST}} \stackrel{\text{def}}{=} \texttt{darr}(\texttt{Type};\alpha.\texttt{S}(\texttt{rec}(\mathit{self}.\texttt{sum}[L_{\texttt{list}}](\\
&\qquad \texttt{Nil} \hookrightarrow \texttt{prod}[\,]();\\
&\qquad \texttt{Cons} \hookrightarrow \texttt{prod}[1;2](1 \hookrightarrow \alpha; 2 \hookrightarrow \mathit{self})))))\\
&L_{\texttt{list}} \stackrel{\text{def}}{=} \texttt{Nil}, \texttt{Cons}\\
&\tau_{\texttt{LIST}} \stackrel{\text{def}}{=} \texttt{prod}[L_{\texttt{list}}](\\
&\qquad \texttt{Nil} \hookrightarrow \texttt{all}\{\texttt{Type}\}(\alpha.\texttt{app}(\mathit{list};\alpha));\\
&\qquad \texttt{Cons} \hookrightarrow \texttt{all}\{\texttt{Type}\}(\alpha.\texttt{parr}(\\
&\qquad\quad \texttt{prod}[1;2](1 \hookrightarrow \alpha; 2 \hookrightarrow \texttt{app}(\mathit{list};\alpha));\\
&\qquad\quad \texttt{app}(\mathit{list};\alpha))))
\end{aligned}
$$

Figure 5.8: The miniVerse$_{\mathbf{P}}$ encoding of the LIST signature

The construction projection form, $\texttt{con}(M)$, allows us to refer to the construction component of M within a construction appearing in M'. The kinding rule for this form is reproduced below:

$$\frac{\Omega \vdash M \ \mathsf{mval} \qquad \Omega \vdash M : \texttt{sig}\{\kappa\}(u.\tau)}{\Omega \vdash \texttt{con}(M) :: \kappa} \tag{C.9o}$$

Similarly, the value projection form, $\texttt{val}(M)$, projects out the value component of M within an expression appearing in M'. The typing rule for this form is reproduced below:

$$\frac{\Omega \vdash M \ \mathsf{mval} \qquad \Omega \vdash M : \texttt{sig}\{\kappa\}(u.\tau)}{\Omega \vdash \texttt{val}(M) : [\texttt{con}(M)/u]\tau} \tag{C.12m}$$

The first premise of both of these rules requires that M be, or stand for, a *module value*, according to the following rules:

$$\frac{}{\Omega \vdash \texttt{struct}(c;e) \ \mathsf{mval}} \tag{C.5a}$$

$$\frac{}{\Omega, X : \sigma \vdash X \ \mathsf{mval}} \tag{C.5b}$$

The reason for this restriction has to do with the *sealing* operation:

$$\frac{\Omega \vdash \sigma \ \mathsf{sig} \qquad \Omega \vdash M : \sigma}{\Omega \vdash \texttt{seal}\{\sigma\}(M) : \sigma} \tag{C.4d}$$

Sealing enforces *representation independence* – the abstract construction components of a sealed module are not treated as equivalent to those of any other sealed module within the program. In other words, sealing is *generative*. The module value restriction above achieves this behavior by simple syntactic means – a sealed module is not a module value, so all sealed modules have to be bound to distinct module variables.

The judgements above obey standard lemmas, including Weakening, Substitution and Decomposition (see Appendix C.1.2.)

We omit certain features of the ML module system in miniVerse$_{\mathbf{P}}$, such as its support for hierarchical modules and functors. Our formulation also does not support "width" subtyping and subkinding for simplicity. These are straightforward extensions of miniVerse$_{\mathbf{P}}$, but because their inclusion would not change the semantics of parametric TSMs, we did not include them (see [61] for a discussion of these features.)

5.2.3 Structural Dynamics

The structural dynamics of modules is defined as a transition system, and is organized around judgements of the following form:

Judgement Form	**Description**
$M \mapsto M'$	M transitions to M'
M val	M is a module value
M $\mathsf{matchfail}$	M raises match failure

The structural dynamics of expressions is also defined as a transition system, and is organized around judgements of the following form:

Judgement Form	**Description**
$e \mapsto e'$	e transitions to e'
e val	e is a value
e $\mathsf{matchfail}$	e raises match failure

We also define auxiliary judgements for *iterated transition*, $e \mapsto^* e'$, and *evaluation*, $e \Downarrow e'$, of expressions.

Definition C.6 (Iterated Transition)**.** *Iterated transition, $e \mapsto^* e'$, is the reflexive, transitive closure of the transition judgement, $e \mapsto e'$.*

Definition C.7 (Evaluation)**.** *$e \Downarrow e'$ iff $e \mapsto^* e'$ and e' val.*

As in previous chapters, our subsequent developments do not make mention of particular rules in the dynamics, so we do not produce these details here. Instead, it suffices to state the following conditions.

The Preservation condition ensures that evaluation preserves typing.

Condition C.10 (Preservation)**.**

1. *If $\vdash M : \sigma$ and $M \mapsto M'$ then $\vdash M : \sigma$.*
2. *If $\vdash e : \tau$ and $e \mapsto e'$ then $\vdash e' : \tau$.*

The Progress condition ensures that evaluation of a well-typed expanded expression cannot "get stuck". We must consider the possibility of match failure in this condition.

Condition C.11 (Progress)**.**

1. *If $\vdash M : \sigma$ then either M val or M $\mathsf{matchfail}$ or there exists an M' such that $M \mapsto M'$.*
2. *If $\vdash e : \tau$ then either e val or e $\mathsf{matchfail}$ or there exists an e' such that $e \mapsto e'$.*

Together, these two conditions constitute the Type Safety Condition.

Sort			Stylized Form	Description
USig	$\hat{\sigma}$	$::=$	$[\![\hat{u} :: \hat{\kappa}; \hat{\tau}]\!]$	signature
UMod	$\hat{M}$	$::=$	$\hat{X}$	module identifier
			$[\![\hat{c}; \hat{e}]\!]$	structure
			$\hat{M} \upharpoonright \hat{\sigma}$	seal
			$(\mathtt{let}\ \hat{X} = \hat{M}\ \mathtt{in}\ \hat{M}) : \hat{\sigma}$	definition
			$\mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for\ expressions\ by\ static}\ e\ \mathtt{in}\ \hat{M}$	peTSM definition
			$\mathtt{let\ syntax}\ \hat{a} = \hat{e}\ \mathtt{for\ expressions\ in}\ \hat{M}$	peTSM binding
			$\mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for\ patterns\ by\ static}\ e\ \mathtt{in}\ \hat{M}$	ppTSM definition
			$\mathtt{let\ syntax}\ \hat{a} = \hat{e}\ \mathtt{for\ patterns\ in}\ \hat{M}$	ppTSM binding

Figure 5.9: Syntax of unexpanded module expressions and signatures in miniVerse$_{\mathbf{P}}$

Sort			Stylized Form	Description
UKind	$\hat{\kappa}$	$::=$	$(\hat{u} :: \hat{\kappa}) \to \hat{\kappa}$	dependent function
			$\langle\!\langle\rangle\!\rangle$	nullary product
			$(\hat{u} :: \hat{\kappa}) \times \hat{\kappa}$	dependent product
			$\mathtt{T}$	type
			$[=\hat{\tau}]$	singleton
UCon	$\hat{c}, \hat{\tau}$	$::=$	$\hat{u}$	construction identifier
			$\hat{t}$	
			$\hat{c} :: \hat{\kappa}$	ascription
			$\lambda \hat{u}.\hat{c}$	abstraction
			$c(c)$	application
			$\langle\!\langle\rangle\!\rangle$	trivial
			$\langle\!\langle \hat{c}, \hat{c} \rangle\!\rangle$	pair
			$\hat{c} \cdot \mathtt{l}$	left projection
			$\hat{c} \cdot \mathtt{r}$	right projection
			$\hat{\tau} \rightharpoonup \hat{\tau}$	partial function
			$\forall(\hat{u} :: \hat{\kappa}).\hat{\tau}$	polymorphic
			$\mu \hat{t}.\hat{\tau}$	recursive
			$\langle \{i \hookrightarrow \hat{\tau}_i\}_{i \in L} \rangle$	labeled product
			$[\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}]$	labeled sum
			$\hat{X} \cdot \mathtt{c}$	construction component

Figure 5.10: Syntax of unexpanded kinds and constructions in miniVerse$_{\mathbf{P}}$

Sort			Stylized Form	Description
UExp	$\hat{e}$	$::=$	$\hat{x}$	identifier
			$\hat{e} : \hat{\tau}$	ascription
			$\mathtt{let\ val}\ \hat{x} = \hat{e}\ \mathtt{in}\ \hat{e}$	value binding
			$\lambda\hat{x}{:}\hat{\tau}.\hat{e}$	abstraction
			$\hat{e}(\hat{e})$	application
			$\Lambda\hat{u}{::}\hat{\kappa}.\hat{e}$	construction abstraction
			$\hat{e}[\hat{c}]$	construction application
			$\mathtt{fold}(\hat{e})$	fold
			$\mathtt{unfold}(\hat{e})$	unfold
			$\langle\{i \hookrightarrow \hat{e}_i\}_{i\in L}\rangle$	labeled tuple
			$\hat{e} \cdot \ell$	projection
			$\mathtt{inj}[\ell](\hat{e})$	injection
			$\mathtt{match}\ \hat{e}\ \{\hat{r}_i\}_{1\leq i\leq n}$	match
			$\hat{X} \cdot \mathtt{v}$	value component
			$\hat{e}$ ‘b‘	peTSM application
URule	$\hat{r}$	$::=$	$\hat{p} \Rightarrow \hat{e}$	match rule
UPat	$\hat{p}$	$::=$	$\hat{x}$	identifier pattern
			$\_$	wildcard pattern
			$\mathtt{fold}(\hat{p})$	fold pattern
			$\langle\{i \hookrightarrow \hat{p}_i\}_{i\in L}\rangle$	labeled tuple pattern
			$\mathtt{inj}[\ell](\hat{p})$	injection pattern
			$\hat{e}$ ‘b‘	ppTSM application

Figure 5.11: Syntax of unexpanded expressions, rules and patterns in miniVerse$_\mathbf{P}$

Sort			Stylized Form	Description
UMType	$\hat{\rho}$	$::=$	$\hat{\tau}$	type annotation
			$\forall\hat{X}{:}\hat{\sigma}.\hat{\rho}$	module parameterization
UMExp	$\hat{\epsilon}$	$::=$	$\hat{a}$	TSM identifier reference
			$\Lambda\hat{X}{:}\hat{\sigma}.\hat{\epsilon}$	module abstraction
			$\hat{\epsilon}(\hat{X})$	module application

Figure 5.12: Syntax of unexpanded TSM types and expressions in miniVerse$_\mathbf{P}$

Sort			Operational Form	Description
MType	ρ	$::=$	$\texttt{type}(\tau)$	type annotation
			$\texttt{allmods}\{\sigma\}(X.\rho)$	module parameterization
MExp	ϵ	$::=$	$\texttt{defref}[a]$	TSM definition reference
			$\texttt{absmod}\{\sigma\}(X.\epsilon)$	module abstraction
			$\texttt{apmod}\{M\}(\epsilon)$	module application

Figure 5.13: Syntax of TSM types and expressions in $\mathsf{miniVerse}_{\mathbf{P}}$

5.2.4 Syntax of the Unexpanded Language

The syntax of the unexpanded language is defined in Figures 5.9 through 5.13.

Each expanded form, with three exceptions, has a corresponding unexpanded form. We refer to these as the *common forms*. The correspondence is defined in Appendix C.2.1.

Kind variables, k, are one exception. Kind variables are used only in the metatheory.

The other two exceptions are constructions of the form $\texttt{con}(M)$ and expressions of the form $\texttt{val}(M)$ where M is of the form $\texttt{struct}(c;e)$. Projection out of a module expression of the form $\texttt{struct}(c;e)$ was supported in the XL only because this is needed to give the language a conventional structural dynamics. Programmers refer to modules exclusively through module identifiers in unexpanded programs.

In addition to the common forms, there are several forms related to pTSMs, highlighted in yellow in these figures. We need syntax for unexpanded TSM types, $\hat{\rho}$, and unexpanded TSM expressions, $\hat{\epsilon}$, to support parameterization and parameter application. Internally, these expand to TSM expressions, ϵ, and TSM types, ρ, respectively.

There is also a context-free textual syntax for the UL. For our purposes, we need only posit the existence of partial metafunctions that satisfy the following condition.

Condition C.12 (Textual Representability)**.** *All of the following must hold:*

1. *For each $\hat{\kappa}$, there exists b such that* $\mathsf{parseUKind}(b) = \hat{\kappa}$.
2. *For each $\hat{c}$, there exists b such that* $\mathsf{parseUCon}(b) = \hat{c}$.
3. *For each $\hat{e}$, there exists b such that* $\mathsf{parseUExp}(b) = \hat{e}$.
4. *For each $\hat{p}$, there exists b such that* $\mathsf{parseUPat}(b) = \hat{p}$.

5.2.5 Typed Expansion

Typed expansion is defined by six groups of judgements. In these judgements, *unexpanded unified contexts*, $\hat{\Omega}$, take the form $\langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle$, where $\mathcal{M}$ is a *module identifier expansion context*, $\mathcal{D}$ is a *construction identifier expansion context*, $\mathcal{G}$ is an *expression identifier expansion context* and Ω is a unified context. Identifier expansion contexts are defined in Appendix C.2.2 and conceptually operate as described in Sec. 3.2, mapping identifiers to variables.

The first group of judgements defines signature and module expansion.

Judgement Form	Description
$\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma\ \mathsf{sig}$	$\hat{\sigma}$ has well-formed expansion σ
$\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma$	$\hat{M}$ has expansion M matching σ

The second group of judgements defines kind and construction expansion.

Judgement Form	Description
$\hat{\Omega} \vdash \hat{\kappa} \rightsquigarrow \kappa \ \mathsf{kind}$	$\hat{\kappa}$ has well-formed expansion κ
$\hat{\Omega} \vdash \hat{c} \rightsquigarrow c :: \kappa$	$\hat{c}$ has expansion c of kind κ

The third group of judgements defines expression, rule and pattern expansion.

Judgement Form	Description
$\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau$	$\hat{e}$ has expansion e of type τ
$\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \rightsquigarrow r : \tau \Mapsto \tau'$	$\hat{r}$ has expansion r taking values of type τ to values of type τ'
$\hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\!\mid \hat{\Omega}'$	$\hat{p}$ has expansion p matching at τ generating hypotheses $\hat{\Omega}'$

The judgements above are defined by the rules given in Appendix C.2.2. Most of these rules simply serve to "mirror" corresponding rules in the statics of the XL, as was described in Sec. 3.2. The interesting rules, governing the forms highlighted in yellow, will be reproduced as we discuss them below.

The remaining judgements assign meaning to TSM types and expressions. We will detail these below. In particular, the fourth group of judgements define TSM type and expression expansion.

Judgement Form	Description
$\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho \ \mathsf{tsmty}$	$\hat{\rho}$ has well-formed expansion ρ
$\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{\epsilon} \rightsquigarrow \epsilon \,@\, \rho$	$\hat{\epsilon}$ has peTSM expression expansion ϵ at ρ
$\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Psi}} \hat{\epsilon} \rightsquigarrow \epsilon \,@\, \rho$	$\hat{\epsilon}$ has ppTSM expression expansion ϵ at ρ

The fifth group of judgements define the statics of TSM expressions.

Judgement Form	Description
$\Omega \vdash \rho \ \mathsf{tsmty}$	ρ is a TSM type
$\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \,@\, \rho$	ϵ is a peTSM expression at ρ
$\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \,@\, \rho$	ϵ is a ppTSM expression at ρ

The sixth group of judgements define the dynamics of TSM expressions.

Judgement Form	Description
$\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto \epsilon'$	peTSM expression ϵ transitions to ϵ'
$\Omega \vdash^{\mathsf{Pat}}_{\Psi} \epsilon \mapsto \epsilon'$	ppTSM expression ϵ transitions to ϵ'
$\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \ \mathsf{normal}$	ϵ is a normal peTSM expression
$\Omega \vdash^{\mathsf{Pat}}_{\Psi} \epsilon \ \mathsf{normal}$	ϵ is a normal ppTSM expression

We define the multi-step transition judgements $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto^* \epsilon'$ and $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mapsto^* \epsilon'$ as the reflexive transitive closures of the corresponding transition judgements. We also define the peTSM expression normalization judgement $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \Downarrow \epsilon'$ iff $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto^* \epsilon'$ and $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon' \ \mathsf{normal}$. Similarly, we define the ppTSM expression normalization judgement $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \Downarrow \epsilon'$ iff $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mapsto^* \epsilon'$ and $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon' \ \mathsf{normal}$.

5.2.6 TSM Definitions

TSMs are scoped to module expressions. (Adding support for TSM definitions scoped to a single expression would be a straightforward exercise, so we omit the details for simplicity.)

peTSM Definitions

The rule governing peTSM definitions is reproduced below:

$$\frac{\begin{array}{cc}\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho\ \mathsf{tsmty} & \emptyset \vdash e_{\text{parse}} : \mathtt{parr}(\mathtt{Body}; \mathtt{ParseResult}(\mathtt{PPrExpr})) \\ e_{\text{parse}} \Downarrow e'_{\text{parse}} & \hat{\Omega} \vdash_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \mathtt{defref}[a]; \Psi, a \hookrightarrow \mathtt{petsm}(\rho; e'_{\text{parse}}) \rangle; \hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma\end{array}}{\hat{\Omega} \vdash_{\langle \mathcal{A}; \Psi \rangle; \hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for}\ \mathtt{expressions}\ \mathtt{by}\ \mathtt{static}\ e_{\text{parse}}\ \mathtt{in}\ \hat{M} \rightsquigarrow M : \sigma} \quad \text{(C.16f)}$$

peTSM definitions differ from ueTSM definitions in that the unexpanded type annotation is an *unexpanded TSM type*, $\hat{\rho}$, rather than an unexpanded type, $\hat{\tau}$. This unexpanded TSM type determines the parameterization of the TSM. The first premise of the rule above expands the unexpanded TSM type to produce a *TSM type*, ρ. The straightforward rules governing TSM type expansion are reproduced below.

$$\frac{\hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau :: \mathsf{Type}}{\hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \mathtt{type}(\tau)\ \mathsf{tsmty}} \quad \text{(C.22a)}$$

$$\frac{\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma\ \mathsf{sig} \qquad \hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma \vdash \hat{\rho} \rightsquigarrow \rho\ \mathsf{tsmty}}{\hat{\Omega} \vdash \forall \hat{X}{:}\hat{\sigma}.\hat{\rho} \rightsquigarrow \mathtt{allmods}\{\sigma\}(X.\rho)\ \mathsf{tsmty}} \quad \text{(C.22b)}$$

Rule (C.22a) defines quantification over modules matching a given signature. There is no mechanism for quantification over types in the calculus because it can be understood as quantification over a module with a single type component.

The second premise of Rule (C.16f) checks that the parse function is of the appropriate type. The types `Body` and `ParseResult(PPrExpr)` are characterized in Appendix C.2.2. The type `PPrExpr` classifies *encodings of parameterized proto-expressions*, which we will return to when we discuss TSM application below.

The third premise of Rule (C.16f) evaluates the parse function to a value.

The final premise of Rule (C.16f) extends the *peTSM context*, $\hat{\Psi}$, which consists of a *TSM identifier expansion context*, $\mathcal{A}$, and a *peTSM definition context*, Ψ. A peTSM definition context maps TSM names, a, to an expanded peTSM definition, $a \hookrightarrow \mathtt{petsm}(\rho; e_{\text{parse}})$, where ρ is the TSM type determined from the annotation and e_{parse} is its parse function. A TSM identifier context maps TSM identifiers, $\hat{a}$, to *TSM expressions*, ϵ. In this case, the TSM expression is simply a reference to the newly introduced TSM definition, $\mathtt{defref}[a]$. We discuss the other TSM expression forms when we discuss TSM abbreviations below.

ppTSM Definitions

The rule governing ppTSM definitions is similar, and is reproduced below:

$$\frac{\begin{array}{cc}\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho \;\mathsf{tsmty} & \emptyset \vdash e_{\text{parse}} : \mathtt{parr(Body;ParseResult(PPrPat))} \\ e_{\text{parse}} \Downarrow e'_{\text{parse}} & \hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \mathtt{defref}[a];\Phi, a \hookrightarrow \mathtt{pptsm}(\rho;e'_{\text{parse}})\rangle} \hat{M} \rightsquigarrow M : \sigma\end{array}}{\hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A};\Phi\rangle} \mathtt{syntax}\; \hat{a}\; \mathtt{at}\; \hat{\rho}\; \mathtt{for\; patterns\; by\; static}\; e_{\text{parse}}\; \mathtt{in}\; \hat{M} \rightsquigarrow M : \sigma} \tag{C.16h}$$

This rule differs from Rule (C.16f) in the type of the parse function and in the fact that the *ppTSM context*, $\hat{\Phi}$, rather than the peTSM context, is updated.

5.2.7 TSM Abbreviations

It is possible to abbreviate a complex TSM expression by binding it to a TSM identifier.

peTSM Abbreviations

The rule governing peTSM abbreviations is reproduced below:

$$\frac{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\langle \mathcal{A};\Psi\rangle} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \rho \qquad \hat{\Omega} \vdash_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \epsilon;\Psi\rangle;\hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma}{\hat{\Omega} \vdash_{\langle \mathcal{A};\Psi\rangle;\hat{\Phi}} \mathtt{let\; syntax}\; \hat{a} = \hat{\epsilon}\; \mathtt{for\; expressions\; in}\; \hat{M} \rightsquigarrow M : \sigma} \tag{C.16g}$$

Here, $\hat{\epsilon}$ is an *unexpanded TSM expression*. The first premise of the rule above expands it, producing a TSM expression ϵ at TSM type ρ. The second premise updates the peTSM identifier expansion context with this TSM expression.

The rules below govern peTSM expression expansion. The first rule handles the base case, when the unexpanded TSM expression is a TSM identifier, $\hat{a}$, by looking it up in $\mathcal{A}$ and determining its TSM type according to the TSM expression typing judgement, $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} \rho$ (which mirrors the rules below, and is defined in Appendix C.2.2.)

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} \rho}{\langle \mathcal{M};\mathcal{D};\mathcal{G};\Omega\rangle \vdash^{\mathsf{Exp}}_{\langle \mathcal{A},\hat{a} \hookrightarrow \epsilon;\Psi\rangle} \hat{a} \rightsquigarrow \epsilon \mathbin{@} \rho} \tag{C.23a}$$

The following rule allows a peTSM expression to itself abstract over a module. (This is necessary to support abbreviated application of parameters other than the first.)

$$\frac{\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma \;\mathsf{sig} \qquad \hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \rho}{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \Lambda \hat{X}{:}\hat{\sigma}.\hat{\epsilon} \rightsquigarrow \mathtt{absmod}\{\sigma\}(X.\epsilon) \mathbin{@} \mathtt{allmods}\{\sigma\}(X.\rho)} \tag{C.23b}$$

The final rule defines the semantics of parameter application.

$$\frac{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \mathtt{allmods}\{\sigma\}(X'.\rho) \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{X} \rightsquigarrow X : \sigma}{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{\epsilon}(\hat{X}) \rightsquigarrow \mathtt{apmod}\{X\}(\epsilon) \mathbin{@} [X/X']\rho} \tag{C.23c}$$

ppTSM Abbreviations

The rule governing ppTSM abbreviations is analagous:

$$\frac{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\langle \mathcal{A};\Phi\rangle} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \rho \qquad \hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \epsilon;\Phi\rangle} \hat{M} \rightsquigarrow M : \sigma}{\hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A};\Phi\rangle} \texttt{let syntax}\ \hat{a} = \hat{\epsilon}\ \texttt{for patterns in}\ \hat{M} \rightsquigarrow M : \sigma} \tag{C.16i}$$

The ppTSM expression expansion judgement appearing as the first premise is defined analagously to the peTSM expression expansion judgement defined above, differing only in that the rule for TSM identifiers consults the ppTSM context rather than the peTSM context. The rules are reproduced in Appendix C.2.2.

5.2.8 TSM Application

peTSM Application

The rule for applying an unexpanded peTSM expression $\hat{\epsilon}$ to a generalized literal form with body b is reproduced below:

$$\frac{\begin{array}{c} \hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\text{app}} \rangle \qquad \hat{\Psi} = \langle \mathcal{A}; \Psi \rangle \\ \hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \texttt{type}(\tau_{\text{final}}) \qquad \Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \Downarrow \epsilon_{\text{normal}} \\ \mathsf{tsmdef}(\epsilon_{\text{normal}}) = a \qquad \Psi = \Psi', a \hookrightarrow \texttt{petsm}(\rho; e_{\text{parse}}) \\ b \downarrow_{\mathsf{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj[SuccessE]}(e_{\text{pproto}}) \qquad e_{\text{pproto}} \uparrow_{\mathsf{PPrExpr}} \grave{e} \\ \Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \dot{e} \looparrowright_{\epsilon_{\text{normal}}} \grave{e}\ ?\ \texttt{type}(\tau_{\text{proto}}) \dashv \omega : \Omega_{\text{params}} \\ \mathsf{seg}(\grave{e})\ \mathsf{segments}\ b \qquad \Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}};\hat{\Omega};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e : \tau_{\text{proto}} \end{array}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{\epsilon}\ \text{‘}b\text{‘} \rightsquigarrow [\omega]e : [\omega]\tau_{\text{proto}}} \tag{C.19p}$$

The first two premises simply deconstruct $\hat{\Omega}$ and $\hat{\Psi}$. Next, we expand $\hat{\epsilon}$ according to the unexpanded peTSM expression expansion rules that we already described above. The resulting TSM expression, ϵ, must be defined at a type (i.e. no quantification must remain.)

The fourth premise performs *peTSM expression normalization*. Normalization is defined in terms of a simple structural dynamics with two stepping rules:

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto \epsilon'}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \texttt{apmod\{}X\texttt{\}(}\epsilon\texttt{)} \mapsto \texttt{apmod\{}X\texttt{\}(}\epsilon'\texttt{)}} \tag{C.29a}$$

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \texttt{apmod\{}X\texttt{\}(absmod\{}\sigma\texttt{\}(}X'.\epsilon\texttt{))} \mapsto [X/X']\epsilon} \tag{C.29b}$$

The peTSM expression normal forms are defined as follows:

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi, a \hookrightarrow \texttt{petsm}(\rho; e_{\text{parse}})} \texttt{defref[}a\texttt{]}\ \mathsf{normal}} \tag{C.35a}$$

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{absmod\{}\sigma\mathtt{\}(}X.\epsilon\mathtt{)} \text{ normal}} \tag{C.35b}$$

$$\frac{\epsilon \neq \mathtt{absmod\{}\sigma\mathtt{\}(}X'.\epsilon'\mathtt{)} \qquad \Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \text{ normal}}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{apmod\{}X\mathtt{\}(}\epsilon\mathtt{)} \text{ normal}} \tag{C.35c}$$

Normalization leaves only those parameter applications that cannot be reduced away immediately, i.e. those specified by the original TSM definition.

The TSM definition at the root of the normalized TSM expression is extracted by the third row of premises in Rule (C.19p). The first of these appeals to the following metafunction to produce the TSM definition's name.

$$\mathsf{tsmdef}(\mathtt{defref[}a\mathtt{]}) = a \tag{C.28a}$$

$$\mathsf{tsmdef}(\mathtt{absmod\{}\sigma\mathtt{\}(}X.\epsilon\mathtt{)}) = \mathsf{tsmdef}(\epsilon) \tag{C.28b}$$

$$\mathsf{tsmdef}(\mathtt{apmod\{}X\mathtt{\}(}\epsilon\mathtt{)}) = \mathsf{tsmdef}(\epsilon) \tag{C.28c}$$

The second premise on the third row then looks up this name within Ψ.

The fourth row of premises in Rule (C.19p) 1) encode the body as a value of the type `Body`; 2) apply the parse function; and 3) decode the result, producing a *parameterized proto-expression*, $\dot{e}$. Parameterized proto-expressions, $\dot{e}$, are ABTs that serve to introduce the parameter bindings into a proto-expression, $\grave{e}$. The operational and stylized syntax of parameterized proto-expression is given in Figure 5.14.

Sort			**Operational Form**	**Stylized Form**	**Description**
PPrExpr	$\dot{e}$	$::=$	$\mathtt{prexp(}\grave{e}\mathtt{)}$	$\grave{e}$	proto-expression
			$\mathtt{prbindmod(}X.\dot{e}\mathtt{)}$	$\Lambda X.\dot{e}$	module binding

Figure 5.14: Syntax of parameterized proto-expressions in miniVerse$_{\mathbf{P}}$

There must be one binder in $\dot{e}$ for each TSM parameter specified by $\mathsf{tsmdef}(\epsilon_{\text{normal}})$. (VerseML inserts 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 (C.19p) 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 reproduced below:

$$\frac{}{\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi, a \hookrightarrow \mathtt{petsm(}\rho;e_{\text{parse}}\mathtt{)}} \mathtt{prexp(}\grave{e}\mathtt{)} \looparrowright_{\mathtt{defref[}a\mathtt{]}} \grave{e} \mathrel{?} \rho \dashv \emptyset : \emptyset} \tag{C.37a}$$

$$\frac{\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \dot{e} \looparrowright_{\epsilon} \grave{e} \mathrel{?} \mathtt{allmods\{}\sigma\mathtt{\}(}X.\rho\mathtt{)} \dashv \omega : \Omega \qquad X \notin \text{dom}(\Omega_{\text{app}})}{\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{prbindmod(}X.\dot{e}\mathtt{)} \looparrowright_{\mathtt{apmod\{}X'\mathtt{\}(}\epsilon\mathtt{)}} \grave{e} \mathrel{?} \rho \dashv (\omega, X'/X) : (\Omega, X : \sigma)} \tag{C.37b}$$

This judgement can be pronounced "when applying peTSM ϵ, $\dot{e}$ has deparameterization $\grave{e}$ leaving ρ with parameter substitution ω". Notice from Rule (C.37b) 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 in Rule (C.19p) performs proto-expansion validation. This involves first checking that the segmentation of $\grave{e}$ is valid, and then checking that the proto-expansion is well-typed 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 it was checked against.

ppTSM Application

The rule governing ppTSM application is similar:

$$\frac{\begin{array}{c}\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\text{app}} \rangle \qquad \hat{\Phi} = \langle \mathcal{A}; \Phi \rangle \\ \hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \hat{\epsilon} \leadsto \epsilon \mathbin{@} \mathtt{type}(\tau_{\text{final}}) \qquad \Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \Downarrow \epsilon_{\text{normal}} \\ \mathsf{tsmdef}(\epsilon_{\text{normal}}) = a \qquad \Phi = \Phi', a \hookrightarrow \mathtt{pptsm}(\rho; e_{\text{parse}}) \\ b \downarrow_{\mathsf{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \mathtt{inj}[\mathtt{SuccessP}](e_{\text{pproto}}) \qquad e_{\text{pproto}} \uparrow_{\mathsf{PPrPat}} \dot{p} \\ \Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi} \dot{p} \looparrowright_{\epsilon_{\text{normal}}} \grave{p} \mathrel{?} \mathtt{type}(\tau_{\text{proto}}) \dashv \omega : \Omega_{\text{params}} \\ \mathsf{seg}(\grave{p}) \text{ segments } b \qquad \grave{p} \leadsto p : \tau_{\text{proto}} \dashv\!\!|^{\omega:\Omega_{\text{params}}; \hat{\Omega}; \hat{\Phi}; b} \hat{\Omega}'\end{array}}{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{\epsilon}\ \text{`}b\text{`} \leadsto p : [\omega]\tau_{\text{proto}} \dashv\!\!| \hat{\Omega}'} \quad \text{(C.21g)}$$

Although patterns themselves cannot make reference to surrounding bindings, the type annotations on spliced patterns can, so we need the notion of a *parameterized proto-pattern*, $\dot{p}$, and a corresponding deparameterization judgement. The necessary definitions, which are analagous to those given above for peTSMs, are given in Appendix C.2.2.

5.2.9 Syntax of Proto-Expansions

Figure 5.15 defines the syntax of proto-kinds, $\grave{\kappa}$ and proto-constructions, $\grave{c}$. Figure 5.16 defines the syntax of proto-expressions, $\grave{e}$, proto-rules, $\grave{r}$, and proto-patterns, $\grave{p}$. All of these are ABTs.

The mapping from expanded forms to proto-expansion forms is given in Appendix C.3. The only "interesting" forms are the forms for references to spliced unexpanded terms, highlighted in yellow in Figure 5.15 and Figure 5.16.

5.2.10 Proto-Expansion Validation

Proto-expansion validation operates essentially as described in Sec. 3.2.10. It is governed by two groups of judgements. The first group of judgements defines proto-kind and proto-construction validation.

Sort			Operational Form	Stylized Form	Description
PrKind	$\grave{\kappa}$	$::=$	$\mathtt{prdarr}(\grave{\kappa}; u.\grave{\kappa})$	$(u :: \grave{\kappa}) \to \grave{\kappa}$	dependent function
			$\mathtt{prunit}$	$\langle\langle\rangle\rangle$	nullary product
			$\mathtt{prdprod}(\grave{\kappa}; u.\grave{\kappa})$	$(u :: \grave{\kappa}) \times \grave{\kappa}$	dependent product
			$\mathtt{prType}$	$\mathtt{T}$	type
			$\mathtt{prS}(\grave{\tau})$	$[=\grave{\tau}]$	singleton
			$\mathtt{splicedk}[m; n]$	$\mathtt{splicedk}[m; n]$	spliced kind
PrCon	$\grave{c}, \grave{\tau}$	$::=$	u	u	construction variable
			t	t	type variable
			$\mathtt{prabs}(u.\grave{c})$	$\lambda u.\grave{c}$	abstraction
			$\mathtt{prapp}(\grave{c}; \grave{c})$	$\grave{c}(\grave{c})$	application
			$\mathtt{prtriv}$	$\langle\langle\rangle\rangle$	trivial
			$\mathtt{prpair}(\grave{c}; \grave{c})$	$\langle\langle \grave{c}, \grave{c} \rangle\rangle$	pair
			$\mathtt{prprl}(\grave{c})$	$\grave{c} \cdot \mathtt{l}$	left projection
			$\mathtt{prprr}(\grave{c})$	$\grave{c} \cdot \mathtt{r}$	right projection
			$\mathtt{prparr}(\grave{\tau}; \grave{\tau})$	$\grave{\tau} \rightharpoonup \grave{\tau}$	partial function
			$\mathtt{prall}\{\grave{\kappa}\}(u.\grave{\tau})$	$\forall(u :: \grave{\kappa}).\grave{\tau}$	polymorphic
			$\mathtt{prrec}(t.\grave{\tau})$	$\mu t.\grave{\tau}$	recursive
			$\mathtt{prprod}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L})$	$\langle \{i \hookrightarrow \grave{\tau}_i\}_{i \in L} \rangle$	labeled product
			$\mathtt{prsum}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L})$	$[\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}]$	labeled sum
			$\mathtt{prcon}(X)$	$X \cdot \mathtt{c}$	construction component
			$\mathtt{splicedc}[m; n; \grave{\kappa}]$	$\mathtt{splicedc}[m; n; \grave{\kappa}]$	spliced construction

Figure 5.15: Syntax of proto-kinds and proto-constructions in miniVerse$_{\mathbf{P}}$

Sort			Operational Form	Stylized Form	Description
PrExp	$\grave{e}$	$::=$	x	x	variable
			$\texttt{prasc}\{\grave{\tau}\}(\grave{e})$	$\grave{e} : \grave{\tau}$	ascription
			$\texttt{prletval}(\grave{e}; x.\grave{e})$	$\texttt{let val}\ x = \grave{e}\ \texttt{in}\ \grave{e}$	value binding
			$\texttt{prlam}\{\grave{\tau}\}(x.\grave{e})$	$\lambda x{:}\grave{\tau}.\grave{e}$	abstraction
			$\texttt{prap}(\grave{e}; \grave{e})$	$\grave{e}(\grave{e})$	application
			$\texttt{prclam}\{\grave{\kappa}\}(u.\grave{e})$	$\Lambda u{::}\grave{\kappa}.\grave{e}$	construction abstraction
			$\texttt{prcap}\{\grave{c}\}(\grave{e})$	$\grave{e}[\grave{c}]$	construction application
			$\texttt{prfold}(\grave{e})$	$\texttt{fold}(\grave{e})$	fold
			$\texttt{prunfold}(\grave{e})$	$\texttt{unfold}(\grave{e})$	unfold
			$\texttt{prtpl}\{L\}(\{i \hookrightarrow \grave{e}_i\}_{i \in L})$	$\langle\{i \hookrightarrow \grave{e}_i\}_{i \in L}\rangle$	labeled tuple
			$\texttt{prprj}[\ell](\grave{e})$	$\grave{e} \cdot \ell$	projection
			$\texttt{prinj}[\ell](\grave{e})$	$\texttt{inj}[\ell](\grave{e})$	injection
			$\texttt{prmatch}[n](\grave{e}; \{\grave{r}_i\}_{1 \leq i \leq n})$	$\texttt{match}\ \grave{e}\ \{\grave{r}_i\}_{1 \leq i \leq n}$	match
			$\texttt{prval}(X)$	$X \cdot \mathbf{v}$	value component
			$\texttt{splicede}[m; n; \grave{\tau}]$	$\texttt{splicede}[m; n; \grave{\tau}]$	spliced expression
PrRule	$\grave{r}$	$::=$	$\texttt{prrule}(p.\grave{e})$	$p \Rightarrow \grave{e}$	rule
PrPat	$\grave{p}$	$::=$	$\texttt{prwildp}$	$\_$	wildcard pattern
			$\texttt{prfoldp}(p)$	$\texttt{fold}(p)$	fold pattern
			$\texttt{prtplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i \in L})$	$\langle\{i \hookrightarrow \grave{p}_i\}_{i \in L}\rangle$	labeled tuple pattern
			$\texttt{prinjp}[\ell](\grave{p})$	$\texttt{inj}[\ell](\grave{p})$	injection pattern
			$\texttt{prval}(X)$	$X \cdot \mathbf{v}$	value component
			$\texttt{splicedp}[m; n; \grave{\tau}]$	$\texttt{splicedp}[m; n; \grave{\tau}]$	spliced pattern

Figure 5.16: Syntax of proto-expressions, proto-rules and proto-patterns in miniVerse$_{\mathbf{P}}$

Judgement Form	Description
$\Omega \vdash^{\mathbb{C}} \grave{\kappa} \rightsquigarrow \kappa$ kind	$\grave{\kappa}$ has well-formed expansion κ
$\Omega \vdash^{\mathbb{C}} \grave{c} \rightsquigarrow c :: \kappa$	$\grave{c}$ has expansion c of kind κ

The second group of judgements defines proto-expression, proto-rule and proto-pattern validation.

Judgement Form	Description
$\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau$	$\grave{e}$ has expansion e of type τ
$\Omega \vdash^{\mathbb{E}} \grave{r} \rightsquigarrow r : \tau \Mapsto \tau'$	$\grave{r}$ has expansion r taking values of type τ to values of type τ'
$\grave{p} \rightsquigarrow p : \tau \dashv\vert^{\mathbb{P}} \hat{\Omega}$	$\grave{p}$ has expansion p matching against τ generating hypotheses $\hat{\Omega}$

Expression splicing scenes, $\mathbb{E}$, are of the form $\omega : \Omega_{\text{params}}; \hat{\Omega}; \hat{\Psi}; \hat{\Phi}; b$, *construction splicing scenes*, $\mathbb{C}$, are of the form $\omega : \Omega_{\text{params}}; \hat{\Omega}; b$, and *pattern splicing scenes*, $\mathbb{P}$, are of the form $\omega : \Omega_{\text{params}}; \hat{\Omega}; \hat{\Phi}; b$. Their purpose is to "remember", during proto-expansion validation, the contexts and literal bodies from the TSM application site (cf. Rules (C.19p) and (C.21g) above), because these are necessary to validate references to spliced terms. They also keep around the parameter substitution and corresponding context, $\omega : \Omega_{\text{params}}$, because type/kind annotations on spliced terms need to be able to access parameters (but not expansion-local bindings.) We write $\mathsf{cs}(\mathbb{E})$ for the construction splicing scene constructed by dropping the TSM contexts from $\mathbb{E}$:

$$\mathsf{cs}(\omega : \Omega_{\text{params}}; \hat{\Omega}; \hat{\Psi}; \hat{\Phi}; b) = \omega : \Omega_{\text{params}}; \hat{\Omega}; b$$

The rules governing references to spliced terms are reproduced below:

$$\frac{\mathsf{parseUKind}(\mathsf{subseq}(b; m; n)) = \hat{\kappa} \qquad \hat{\Omega} \vdash \hat{\kappa} \rightsquigarrow \kappa \;\mathsf{kind} \qquad \hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\text{app}} \rangle \qquad \mathsf{dom}(\Omega) \cap \mathsf{dom}(\Omega_{\text{app}}) = \emptyset}{\Omega \vdash^{\omega:\Omega_{\text{params}}; \hat{\Omega}; b} \mathtt{splicedk}[m; n] \rightsquigarrow \kappa \;\mathsf{kind}} \tag{C.39f}$$

$$\frac{\mathbb{C} = \omega : \Omega_{\text{params}}; \hat{\Omega}; b \qquad \Omega_{\text{params}} \vdash^{\mathbb{C}} \grave{\kappa} \rightsquigarrow \kappa \;\mathsf{kind} \qquad \mathsf{parseUCon}(\mathsf{subseq}(b; m; n)) = \hat{c} \qquad \hat{\Omega} \vdash \hat{c} \rightsquigarrow c :: [\omega]\kappa \qquad \hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\text{app}} \rangle \qquad \mathsf{dom}(\Omega) \cap \mathsf{dom}(\Omega_{\text{app}}) = \emptyset}{\Omega \vdash^{\mathbb{C}} \mathtt{splicedc}[m; n; \grave{\kappa}] \rightsquigarrow c :: \kappa} \tag{C.40p}$$

$$\frac{\mathbb{E} = \omega : \Omega_{\text{params}}; \hat{\Omega}; \hat{\Psi}; \hat{\Phi}; b \qquad \Omega_{\text{params}} \vdash^{\mathsf{cs}(\mathbb{E})} \grave{\tau} \rightsquigarrow \tau :: \mathsf{Type} \qquad \mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e} \qquad \hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e : [\omega]\tau \qquad \hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\text{app}} \rangle \qquad \mathsf{dom}(\Omega) \cap \mathsf{dom}(\Omega_{\text{app}}) = \emptyset}{\Omega \vdash^{\mathbb{E}} \mathtt{splicede}[m; n; \grave{\tau}] \rightsquigarrow e : \tau} \tag{C.41p}$$

$$\frac{\Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}}; \hat{\Omega}; b} \grave{\tau} \rightsquigarrow \tau :: \mathsf{Type} \qquad \mathsf{parseUPat}(\mathsf{subseq}(b; m; n)) = \hat{p} \qquad \hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : [\omega]\tau \dashv\vert \hat{\Omega}'}{\mathtt{splicedp}[m; n; \grave{\tau}] \rightsquigarrow p : \tau \dashv\vert^{\omega:\Omega_{\text{params}}; \hat{\Omega}; \hat{\Phi}; b} \hat{\Omega}'} \tag{C.43e}$$

Notice that the kind/type annotations on spliced terms can refer to the provided parameters, but not to bindings local to the expansion. The parameter substitution, ω, must be applied after expanding the annotations because the parameter names are not bound at the application site.

5.2.11 Metatheory

A more detailed account of the metatheory is given in Appendix C.4. We will summarize the key theorems below.

TSM Expression Evaluation

The following theorems establish a notion of TSM type safety based on preservation and progress for TSM expression evaluation.

Theorem C.27 (peTSM Preservation). *If* $\Omega \vdash_{\Psi}^{\mathsf{Exp}} \epsilon \mathbin{@} \rho$ *and* $\Omega \vdash_{\Psi}^{\mathsf{Exp}} \epsilon \mapsto \epsilon'$ *then* $\Omega \vdash_{\Psi}^{\mathsf{Exp}} \epsilon' \mathbin{@} \rho$.

Theorem C.30 (ppTSM Preservation). *If* $\Omega \vdash_{\Phi}^{\mathsf{Pat}} \epsilon \mathbin{@} \rho$ *and* $\Omega \vdash_{\Phi}^{\mathsf{Pat}} \epsilon \mapsto \epsilon'$ *then* $\Omega \vdash_{\Phi}^{\mathsf{Pat}} \epsilon' \mathbin{@} \rho$.

Theorem C.33 (peTSM Progress). *If* $\Omega \vdash_{\Psi}^{\mathsf{Exp}} \epsilon \mathbin{@} \rho$ *then either* $\Omega \vdash_{\Psi}^{\mathsf{Exp}} \epsilon \mapsto \epsilon'$ *for some* ϵ' *or* $\Omega \vdash_{\Psi}^{\mathsf{Exp}} \epsilon\ \mathsf{normal}$.

Theorem C.34 (ppTSM Progress). *If* $\Omega \vdash_{\Phi}^{\mathsf{Pat}} \epsilon \mathbin{@} \rho$ *then either* $\Omega \vdash_{\Phi}^{\mathsf{Pat}} \epsilon \mapsto \epsilon'$ *for some* ϵ' *or* $\Omega \vdash_{\Phi}^{\mathsf{Pat}} \epsilon\ \mathsf{normal}$.

Typed Expansion

There are also a number of theorems that establish that typed expansion generates a well-typed expansion.

The top-level theorem is the typed expansion theorem for modules.

Theorem C.44 (Module Expansion). *If* $\langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma$ *then* $\Omega \vdash M : \sigma$.

(The proof of this theorem requires proving the corresponding theorems about the other typed expansion judgements, as well as the proto-expansion validation judgements – see Appendix C.4.)

peTSM Abstract Reasoning Principles

The following theorem summarizes the abstract reasoning principles available to programmers when applying a peTSM. Descriptions of labeled clauses are given inline.

Theorem C.47 (peTSM Abstract 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)*** $\hat{\Omega} \vdash_{\hat{\Psi}}^{\mathsf{Exp}} \hat{e} \rightsquigarrow \epsilon \mathbin{@} \mathtt{type}(\tau')$ *and* $\Omega_{app} \vdash e : \tau'$ *for* τ' *such that* $\Omega_{app} \vdash \tau' <: \tau$

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

4. $\Omega_{app} \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \Downarrow \epsilon_{normal}$
5. $\mathsf{tsmdef}(\epsilon_{normal}) = a$
6. $\Psi = \Psi', a \hookrightarrow \texttt{petsm}(\rho; e_{parse})$
7. $b \downarrow_{\mathsf{Body}} e_{body}$
8. $e_{parse}(e_{body}) \Downarrow \texttt{inj}[\texttt{SuccessE}]\,\texttt{(}e_{pproto}\texttt{)}$
9. $e_{pproto} \uparrow_{\mathsf{PPrExpr}} \dot{e}$
10. $\Omega_{app} \vdash^{\mathsf{Exp}}_{\Psi} \dot{e} \looparrowright_{\epsilon_{normal}} \grave{e} \; ? \; \texttt{type(}\tau_{proto}\texttt{)} \dashv \omega : \Omega_{params}$
11. ***(Segmentation)*** $\mathsf{seg}(\grave{e})$ segments b

    The segmentation determined by the proto-expansion actually segments the literal body (i.e. each segment is in-bounds and the segments are non-overlapping.)
12. $\Omega_{params} \vdash^{\omega:\Omega_{params}; \hat{\Omega}; \hat{\Psi}; \hat{\Phi}; b} \grave{e} \leadsto e' : \tau_{proto}$
13. $e = [\omega]e'$
14. $\tau = [\omega]\tau_{proto}$
15. $\mathsf{summary}(\grave{e}) = \{\texttt{splicedk}[m_i; n_i]\}_{0 \le i < n_{kind}} \cup \{\texttt{splicedc}[m_i'; n_i'; \grave{\kappa}_i']\}_{0 \le i < n_{con}} \cup \{\texttt{splicede}[m_i''; n_i''; \grave{\tau}_i]\}_{0 \le i < n_{exp}}$
16. ***(Kinding 1)*** $\{\hat{\Omega} \vdash \mathsf{parseUKind}(\mathsf{subseq}(b; m_i; n_i)) \leadsto \kappa_i \;\mathsf{kind}\}_{0 \le i < n_{kind}}$ *and* $\{\Omega_{app} \vdash \kappa_i \;\mathsf{kind}\}_{0 \le i < n_{kind}}$

    Each spliced kind has a well-formed expansion at the application site.
17. ***(Kinding 2)*** $\{\Omega_{params} \vdash^{\omega:\Omega_{params}; \hat{\Omega}; b} \grave{\kappa}_i' \leadsto \kappa_i' \;\mathsf{kind}\}_{0 \le i < n_{con}}$ *and* $\{\Omega_{app} \vdash [\omega]\kappa_i' \;\mathsf{kind}\}_{0 \le i < n_{con}}$

    Each kind annotation on a spliced construction has a well-formed expansion at the application site.
18. ***(Kinding 3)*** $\{\hat{\Omega} \vdash \mathsf{parseUCon}(\mathsf{subseq}(b; m_i'; n_i')) \leadsto c_i :: [\omega]\kappa_i'\}_{0 \le i < n_{con}}$ *and* $\{\Omega_{app} \vdash c_i :: [\omega]\kappa_i'\}_{0 \le i < n_{con}}$

    Each spliced construction is well-kinded consistent with its kind annotation.
19. ***(Kinding 4)*** $\{\Omega_{params} \vdash^{\omega:\Omega_{params}; \hat{\Omega}; b} \grave{\tau}_i \leadsto \tau_i :: \texttt{Type}\}_{0 \le i < n_{exp}}$ *and* $\{\Omega_{app} \vdash [\omega]\tau_i :: \texttt{Type}\}_{0 \le i < n_{exp}}$

    Each type annotation on a spliced expression has a well-formed expansion at the application site.
20. ***(Typing 2)*** $\{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \mathsf{parseUExp}(\mathsf{subseq}(b; m_i''; n_i'')) \leadsto e_i : [\omega]\tau_i\}_{0 \le i < n_{exp}}$ *and* $\{\Omega_{app} \vdash e_i : [\omega]\tau_i\}_{0 \le i < n_{exp}}$

    Each spliced expression is well-typed consistent with its type annotation.
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* $\{x_i\}_{0 \le i < n_{exp}}$

    The final expansion can be decomposed into a term with variables in place of each spliced kind, construction, expression and parameter. The expansions of these spliced kinds, constructions and expressions, as well as the provided parameters, can be substituted into this term in the standard capture avoiding manner.

22. *(**Context Independence**)*

$$\mathsf{fv}(e'') \subset \{k_i\}_{0\leq i<n_{kind}} \cup \{u_i\}_{0\leq i<n_{con}} \cup \{x_i\}_{0\leq i<n_{exp}} \cup dom(\Omega_{params})$$

The decomposed term is independent of the application site context.

ppTSM Abstract Reasoning Principles

The following theorem summarizes the abstract reasoning principles available to programmers when applying a ppTSM. Descriptions of labeled clauses are given inline.

Theorem C.49 (ppTSM Abstract Reasoning Principles)**.** *If* $\hat{\Omega} \vdash_{\hat{\Phi}} \hat{\epsilon}$ ‘b‘ $\rightsquigarrow p : \tau \dashv\!\mid \hat{\Omega}'$ *then:*

1. $\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{app} \rangle$
2. $\hat{\Phi} = \langle \mathcal{A}; \Phi \rangle$
3. *(**Typing 1**)* $\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \hat{\epsilon} \rightsquigarrow \epsilon$ @ $\mathtt{type}(\tau')$ *and* $\Omega_{app} \vdash p : \tau' \dashv\!\mid \hat{\Omega}'$ *for* τ' *such that* $\Omega_{app} \vdash \tau' <: \tau$

   The final expansion matches values of the type specified by the ppTSM's type annotation.
4. $\Omega_{app} \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \Downarrow \epsilon_{normal}$
5. $\mathsf{tsmdef}(\epsilon_{normal}) = a$
6. $\Phi = \Phi', a \hookrightarrow \mathtt{pptsm}(\rho; e_{parse})$
7. $b \downarrow_{\mathsf{Body}} e_{body}$
8. $e_{parse}(e_{body}) \Downarrow \mathtt{inj}[\mathtt{SuccessP}](e_{pproto})$
9. $e_{pproto} \uparrow_{\mathsf{PPrPat}} \dot{p}$
10. $\Omega_{app} \vdash^{\mathsf{Pat}}_{\Phi} \dot{p} \looparrowright_{\epsilon_{normal}} \grave{p}$? $\mathtt{type}(\tau_{proto}) \dashv \omega : \Omega_{params}$
11. *(**Segmentation**)* $\mathsf{seg}(\grave{p})$ 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.)
12. $\grave{p} \rightsquigarrow p : \tau_{proto} \dashv\!\mid^{\omega:\Omega_{params};\hat{\Omega};\hat{\Phi};b} \hat{\Omega}'$
13. $\tau' = [\omega]\tau_{proto}$
14. $\mathsf{summary}(\grave{e}) = \{\mathtt{splicedk}[m_i; n_i]\}_{0\leq i<n_{kind}} \cup \{\mathtt{splicedc}[m'_i; n'_i; \grave{\kappa}'_i]\}_{0\leq i<n_{con}} \cup \{\mathtt{splicedp}[m''_i; n''_i; \grave{\tau}_i]\}_{0\leq i<n_{pat}}$
15. *(**Kinding 1**)* $\{\hat{\Omega} \vdash \mathsf{parseUKind}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow \kappa_i\ \mathsf{kind}\}_{0\leq i<n_{kind}}$ *and* $\{\Omega_{app} \vdash \kappa_i\ \mathsf{kind}\}_{0\leq i<n_{kind}}$

    Each spliced kind has a well-formed expansion at the application site.
16. *(**Kinding 2**)* $\{\Omega_{params} \vdash^{\omega:\Omega_{params};\hat{\Omega};b} \grave{\kappa}'_i \rightsquigarrow \kappa'_i\ \mathsf{kind}\}_{0\leq i<n_{con}}$ *and* $\{\Omega_{app} \vdash [\omega]\kappa'_i\ \mathsf{kind}\}_{0\leq i<n_{con}}$

    Each kind annotation on a spliced construction has a well-formed expansion at the application site.
17. *(**Kinding 3**)* $\{\hat{\Omega} \vdash \mathsf{parseUCon}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow c_i :: [\omega]\kappa'_i\}_{0\leq i<n_{con}}$ *and* $\{\Omega_{app} \vdash c_i :: [\omega]\kappa'_i\}_{0\leq i<n_{con}}$

    Each spliced construction is well-kinded consistent with its kind annotation.
18. *(**Kinding 4**)* $\{\Omega_{params} \vdash^{\omega:\Omega_{params};\hat{\Omega};b} \grave{\tau}_i \rightsquigarrow \tau_i :: \mathtt{Type}\}_{0\leq i<n_{pat}}$ *and* $\{\Omega_{app} \vdash [\omega]\tau_i :: \mathtt{Type}\}_{0\leq i<n_{pat}}$

Each type annotation on a spliced expression has a well-formed expansion at the application site.

19. *(**Typing 2**)* $\{\hat{\Omega} \vdash_{\hat{\Phi}} \mathsf{parseUPat}(\mathsf{subseq}(b; m''_i; n''_i)) \rightsquigarrow p_i : [\omega]\tau_i \dashv\!| \langle \emptyset; \emptyset; \mathcal{G}_i; \Omega_i \rangle\}_{0 \leq i < n_{pat}}$ *and* $\{\Omega_{app} \vdash p_i : [\omega]\tau_i \dashv\!| \Omega_i\}_{0 \leq i < n_{pat}}$

    Each spliced pattern has a well-typed expansion that matches values of the type indicated by the corresponding type annotation in the splice summary.

20. *(**No Hidden Bindings**)* $\hat{\Omega}' = \langle \emptyset; \emptyset; \biguplus_{0 \leq i < n_{pat}} \mathcal{G}_i; \bigcup_{0 \leq i < n_{pat}} \Omega_i \rangle$

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

Chapter 6

Static Evaluation

In the previous chapters, we have assumed that the parse functions in TSM definitions are closed expanded expressions. This is unrealistic in practice – writing a parser generally requires access to various libraries. Moreover, the parse function might itself be written more concisely using TSMs. In this chapter, we address these problems by introducing a *static environment* shared between parse functions.

6.1 Static Values

Figure 6.1 shows an example of a module, `ParserCombos` (see Sec. 2.4.6). The **`static`** qualifier indicates that this module is bound for use within the parse functions of the subsequent TSM 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) =>
    (* ... *) ParserCombos.alt (* ... *)
end

syntax $b at T' by
  static fn(b) =>
    (* ... *) ParserCombos.alt (* ... *)
end

val y = (* ParserCombos CANNOT be used here *)
```

Figure 6.1: Binding a static module for use within parse functions

The values that arise during the the evaluation of parse functions do not need to persist from "compile-time" to "run-time", so we do not need a full staged computation system [119]. Instead, a sequence of static bindings operates like a lexically-scoped read-evaluate-print loop (REPL), in that each static expression is evaluated immediately and the evaluated values are tracked by a *static environment*.

6.2 Applying TSMs Within TSM Definitions

TSMs and TSM 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 particular relevance to TSM providers.

6.2.1 Quasiquotation

TSMs must generate 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 TSMs that provide support for *quasiquotation syntax* (similar to that built in to languages like Lisp [18] and Scala [108]):

```
static syntax $proto_expr at proto_expr by static fn(b) =>
  (* proto-expression quasiquotation parser here *)
end

static syntax $proto_typ at proto_typ by static fn(b) =>
  (* proto-type quasiquotation parser here *)
end
```

For example, the following expression:

```
val gx = $proto_expr `g(x)`
```

is more concise than its expansion:

```
val gx = App(Var 'g', Var 'x')
```

Anti-quotation, i.e. splicing in an expression of type `proto_expr` (or `proto_pat`), is performed by prefixing a variable or parenthesized expression with `%`:

```
val fgx = $proto_expr `f(%gx)`
```

The expansion of this term is:

```
val fgx = App(Var 'f', gx)
```

6.2.2 Grammar-Based Parser Generators

In Sec. 2.4.5, we discussed a number of grammar-based parser generators. Abstractly, a parser generator is a module matching the signature `PARSEGEN` defined in Figure 6.2. Let us assume a module `P : PARSEGEN` and a grammar of spliced unexpanded expressions that have a given type annotation, `spliced_uexp : proto_typ -> P.grammar(proto_expr)`, in the discussion below.

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

Figure 6.2: A signature for parser generators. The type function `parse_result` was defined in Figure 3.3.

Rather than constructing a grammar using various operations (whose specifications are elided in `PARSEGEN`), we wish to use a syntax for grammars that follows standard conventions. We can do so by defining a static parametric TSM `$grammar`:

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

Using these definitions, we might define a TSM for regexes (implementing a subset of the POSIX regex syntax for simplicity) as shown in Figure 6.3.

```
static module RS : RX = (* ... *)
static module RU = RXUtil(RS)
syntax $rx(R : RX) at R.t by static
  P.generate ($grammar P proto_expr {|
    start <- ""
      fn () => $proto_expr `R.Empty`
    start <- "(" start ")"
      fn e => e
    token str_tok
      RU.parse "[^(@$]+" (* cannot use $rx within its own def *)
    start <- str_tok
      fn s => $proto_expr `R.Str %(str_to_proto_strlit s)`
    start <- start start
      fn e1 e2 => $proto_expr `R.Seq (%e1, %e2)`
    start <- start "|" start
      fn e1 e2 => $proto_expr `R.Or (%e1, %e2)`
    start <- start "*"
      fn e => $proto_expr `R.Star %e`

    using spliced_uexp ($proto_typ `R.t`) as spliced_rx
    start <- "%{" spliced_rx "}"
      fn e => e

    using spliced_uexp ($proto_typ `string`) as spliced_str
    start <- "${" spliced_str "}"
      fn e => $proto_expr `R.Str %(e)`
  |})
end
```

Figure 6.3: A grammar-based definition of `$rx`

6.3 Library Management

In the examples above, we explicitly qualified various definitions with the **static** keyword to make them available within static values. This captures the essential nature of the problem of static evaluation, but 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 achieved by a library manager.

For example, a language-external library manager for VerseML similar to SML/NJ's CM [19] could support a **static** qualifier on imported libraries, which would place the definitions exported by the imported library into the static phase of the library being defined. In particular, a library definition in such a compilation manager might look like this:

```
Library
  (* ... exported module, signature and TSM names ... *)
is
  (* ... files defining those exports ... *)

  (* imports: *)
  static parsegen.cm
```

A similar approach could be taken for languages the incorporate library management directly into the syntax of programs, e.g. Scala [93]:

```
static import edu.cmu.comar.parsegen
```

For the sake of generality and simplicity, we will leave the details of library and compilation management out of our formal developments (following the approach taken in the definition of Standard ML [89].) The problem of packaging macros into components has been studied for term-rewriting macros [31].

An alternative design that allows for the explicit lowering of standard-phase bindings to the static phase has been proposed for OCaml [132].

6.4 miniVerse$_{\mathbf{PH}}$

We will now formalize the mechanisms just discussed by developing a reduced calculus, miniVerse$_{\mathbf{PH}}$. This calculus is defined identically to miniVerse$_{\mathbf{P}}$ with the exception of the syntax and semantics of unexpanded module expressions, $\hat{M}$, so we assume all of the definitions that were given in Appendix C without restating them.

6.4.1 Syntax of Unexpanded Modules

The syntax of unexpanded modules is defined in Figure 6.4. The parts of this figure that differ from Figure 5.9 are highlighted in yellow. Each binding form has a *phase* annotation, φ, and parse functions are now unexpanded expressions, $\hat{e}$, rather than expanded expressions, e. In the textual syntax, the phase annotation `standard` is assumed when no phase annotation has been given.

Sort			Stylized Form	Description
Phase	φ	$::=$	`standard`	standard phase
			`static`	static phase
UMod	$\hat{M}$	$::=$	$\hat{X}$	module identifier
			$[\![\hat{c}; \hat{e}]\!]$	structure
			$\hat{M} \upharpoonright \hat{\sigma}$	seal
			$(\varphi\ \texttt{let}\ \hat{X} = \hat{M}\ \texttt{in}\ \hat{M}) : \hat{\sigma}$	definition
			$\varphi\ \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\rho}\ \texttt{for expressions by static}\ \hat{e}\ \texttt{in}\ \hat{M}$	peTSM definition
			$\varphi\ \texttt{let syntax}\ \hat{a} = \hat{e}\ \texttt{for expressions in}\ \hat{M}$	peTSM binding
			$\varphi\ \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\rho}\ \texttt{for patterns by static}\ \hat{e}\ \texttt{in}\ \hat{M}$	ppTSM definition
			$\varphi\ \texttt{let syntax}\ \hat{a} = \hat{e}\ \texttt{for patterns in}\ \hat{M}$	ppTSM binding

Figure 6.4: Syntax of unexpanded modules in miniVerse$_{\mathbf{PH}}$

6.4.2 Module Expansion

The module expansion judgement in miniVerse$_{\mathbf{PH}}$ takes the following form:

Judgement Form	Description
$\hat{\Omega} \vdash^{\Sigma}_{\hat{\Psi};\hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma$	$\hat{M}$ has expansion M matching σ

The difference here is that there is now a *static environment*, Σ. Static environments take the form $\omega : \hat{\Omega}; \hat{\Psi}; \hat{\Phi}$, where ω is a substitution.

The static environment passes opaquely through the subsumption rule and the rules governing module identifiers, structures and sealing:

$$\frac{\hat{\Omega} \vdash^{\Sigma}_{\hat{\Psi};\hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma \qquad \hat{\Omega} \vdash \sigma <: \sigma'}{\hat{\Omega} \vdash^{\Sigma}_{\hat{\Psi};\hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma'} \tag{6.1a}$$

$$\frac{}{\hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma \vdash^{\Sigma}_{\hat{\Psi};\hat{\Phi}} \hat{X} \rightsquigarrow X : \sigma} \tag{6.1b}$$

$$\frac{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c :: \kappa \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : [c/u]\tau}{\hat{\Omega} \vdash^{\Sigma}_{\hat{\Psi};\hat{\Phi}} [\![\hat{c}; \hat{e}]\!] \rightsquigarrow \texttt{struct}(c; e) : \texttt{sig}\{\kappa\}(u.\tau)} \tag{6.1c}$$

$$\frac{\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma\ \mathsf{sig} \qquad \hat{\Omega} \vdash^{\Sigma}_{\hat{\Psi};\hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma}{\hat{\Omega} \vdash^{\Sigma}_{\hat{\Psi};\hat{\Phi}} \hat{M} \upharpoonright \hat{\sigma} \rightsquigarrow \texttt{seal}\{\sigma\}(M) : \sigma} \tag{6.1d}$$

Each binding form in the syntax of $\hat{M}$ is qualified with a *phase*, φ, which is either `standard` or `static`. The static environment passes opaquely through the standard phase module let binding construct:

$$\frac{\hat{\Omega} \vdash^{\Sigma}_{\hat{\Psi};\hat{\Phi}} \hat{M} \leadsto M : \sigma \qquad \hat{\Omega} \vdash \hat{\sigma}' \leadsto \sigma' \text{ sig} \\ \hat{\Omega}, \hat{X} \leadsto X : \sigma \vdash^{\Sigma}_{\hat{\Psi};\hat{\Phi}} \hat{M}' \leadsto M' : \sigma'}{\hat{\Omega} \vdash^{\Sigma}_{\hat{\Psi};\hat{\Phi}} (\texttt{standard let } \hat{X} = \hat{M} \texttt{ in } \hat{M}') : \hat{\sigma}' \leadsto \texttt{mlet\{}\sigma'\texttt{\}(}M; X.M'\texttt{)} : \sigma'} \tag{6.1e}$$

The rule for the `static` phase module let binding construct, on the other hand, calls for the module expression being bound to be evaluated to a module value under the current environment. The substitution and corresponding unexpanded context is then extended with this module value:

$$\frac{\Sigma = \omega : \hat{\Omega}_S; \hat{\Psi}_S; \hat{\Phi}_S \\ \hat{\Omega}_S \vdash^{\Sigma}_{\hat{\Psi}_S;\hat{\Phi}_S} \hat{M} \leadsto M : \sigma \qquad [\omega]M \Downarrow M' \\ \hat{\Omega} \vdash \hat{\sigma}' \leadsto \sigma' \text{ sig} \qquad \hat{\Omega} \vdash^{\omega, M'/X : \hat{\Omega}_S, \hat{X} \leadsto X:\sigma; \hat{\Psi}_S; \hat{\Phi}_S}_{\hat{\Psi};\hat{\Phi}} \hat{M}' \leadsto M' : \sigma'}{\hat{\Omega} \vdash^{\Sigma}_{\hat{\Psi};\hat{\Phi}} (\texttt{static let } \hat{X} = \hat{M} \texttt{ in } \hat{M}') : \hat{\sigma}' \leadsto M' : \sigma'} \tag{6.1f}$$

The standard peTSM definition construct is governed by the following rule:

$$\frac{\hat{\Omega} \vdash \hat{\rho} \leadsto \rho \text{ tsmty} \qquad \Sigma = \omega : \hat{\Omega}_S; \hat{\Psi}_S; \hat{\Phi}_S \\ \hat{\Omega}_S \vdash_{\hat{\Psi}_S;\hat{\Phi}_S} \hat{e}_{\text{parse}} \leadsto e_{\text{parse}} : \texttt{parr(Body;ParseResult(PPrExpr))} \\ [\omega]e_{\text{parse}} \Downarrow e'_{\text{parse}} \qquad \hat{\Omega} \vdash^{\Sigma}_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \texttt{defref[}a\texttt{]}; \Psi, a \hookrightarrow \texttt{petsm(}\rho; e'_{\text{parse}}\texttt{)} \rangle; \hat{\Phi}} \hat{M} \leadsto M : \sigma}{\hat{\Omega} \vdash^{\Sigma}_{\langle \mathcal{A}; \Psi \rangle; \hat{\Phi}} \texttt{standard syntax } \hat{a} \texttt{ at } \hat{\rho} \texttt{ for expressions by static } \hat{e}_{\text{parse}} \texttt{ in } \hat{M} \leadsto M : \sigma} \tag{6.1g}$$

The difference here is that the parse function is an unexpanded (rather than an expanded) expression. It is expanded under the static environment's unified context, $\hat{\Omega}_S$. Then the substitution, ω, is applied to the resulting expanded parse function before it is added to the peTSM context.

The static peTSM definition construct operates similarly, differing only in that the static environment's peTSM context is extended, rather than the standard peTSM context:

$$\frac{\hat{\Omega} \vdash \hat{\rho} \leadsto \rho \text{ tsmty} \qquad \Sigma = \omega : \hat{\Omega}_S; \hat{\Psi}_S; \hat{\Phi}_S \qquad \hat{\Psi}_S = \langle \mathcal{A}_S; \Psi_S \rangle \\ \hat{\Omega}_S \vdash_{\hat{\Psi}_S;\hat{\Phi}_S} \hat{e}_{\text{parse}} \leadsto e_{\text{parse}} : \texttt{parr(Body;ParseResult(PPrExpr))} \\ [\omega]e_{\text{parse}} \Downarrow e'_{\text{parse}} \qquad \hat{\Omega} \vdash^{\omega:\hat{\Omega}_S; \langle \mathcal{A}_S \uplus \hat{a} \hookrightarrow \texttt{defref[}a\texttt{]}; \Psi_S, a \hookrightarrow \texttt{petsm(}\rho; e'_{\text{parse}}\texttt{)} \rangle; \hat{\Phi}_S}_{\hat{\Psi};\hat{\Phi}} \hat{M} \leadsto M : \sigma}{\hat{\Omega} \vdash^{\Sigma}_{\hat{\Psi};\hat{\Phi}} \texttt{static syntax } \hat{a} \texttt{ at } \hat{\rho} \texttt{ for expressions by static } \hat{e}_{\text{parse}} \texttt{ in } \hat{M} \leadsto M : \sigma} \tag{6.1h}$$

The static environment passes opaquely through the standard peTSM abbreviation construct:

$$\frac{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\langle \mathcal{A}; \Psi \rangle} \hat{\epsilon} \leadsto \epsilon \mathrel{@} \rho \qquad \hat{\Omega} \vdash^{\Sigma}_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \epsilon; \Psi \rangle; \hat{\Phi}} \hat{M} \leadsto M : \sigma}{\hat{\Omega} \vdash^{\Sigma}_{\langle \mathcal{A}; \Psi \rangle; \hat{\Phi}} \texttt{standard let syntax } \hat{a} = \hat{\epsilon} \texttt{ for expressions in } \hat{M} \leadsto M : \sigma} \tag{6.1i}$$

The static peTSM abbreviation construct updates the static peTSM identifier expansion context, $\mathcal{A}_S$:

$$\frac{\begin{array}{c}\hat{\Omega} \vdash^{\mathsf{Exp}}_{\langle \mathcal{A};\Psi\rangle} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \rho \\ \Sigma = \omega : \hat{\Omega}_S; \hat{\Psi}_S; \hat{\Phi}_S \qquad \hat{\Psi}_S = \langle \mathcal{A}_S; \Psi_S \rangle \\ \hat{\Omega} \vdash^{\omega:\Omega_S;\langle \mathcal{A}_S \uplus \hat{a} \hookrightarrow \epsilon;\Psi_S\rangle;\hat{\Phi}_S}_{\hat{\Psi};\hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma\end{array}}{\hat{\Omega} \vdash^{\Sigma}_{\hat{\Psi};\hat{\Phi}} \texttt{static let syntax } \hat{a} = \hat{e} \texttt{ for expressions in } \hat{M} \rightsquigarrow M : \sigma} \tag{6.1j}$$

The rules governing ppTSM definitions and abbreviations are analagous:

$$\frac{\begin{array}{c}\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho \ \mathsf{tsmty} \qquad \Sigma = \omega : \hat{\Omega}_S; \hat{\Psi}_S; \hat{\Phi}_S \\ \hat{\Omega}_S \vdash_{\hat{\Psi}_S;\hat{\Phi}_S} \hat{e}_{\text{parse}} \rightsquigarrow e_{\text{parse}} : \texttt{parr(Body;ParseResult(PPrPat))} \\ [\omega]e_{\text{parse}} \Downarrow e'_{\text{parse}} \qquad \hat{\Omega} \vdash^{\Sigma}_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \texttt{defref}[a];\Phi, a \hookrightarrow \texttt{pptsm}(\rho;e'_{\text{parse}})\rangle} \hat{M} \rightsquigarrow M : \sigma\end{array}}{\hat{\Omega} \vdash^{\Sigma}_{\hat{\Psi};\langle \mathcal{A};\Phi\rangle} \texttt{standard syntax } \hat{a} \texttt{ at } \hat{\rho} \texttt{ for patterns by static } \hat{e}_{\text{parse}} \texttt{ in } \hat{M} \rightsquigarrow M : \sigma} \tag{6.1k}$$

$$\frac{\begin{array}{c}\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho \ \mathsf{tsmty} \qquad \Sigma = \omega : \hat{\Omega}_S; \hat{\Psi}_S; \hat{\Phi}_S \qquad \hat{\Phi}_S = \langle \mathcal{A}_S; \Phi_S \rangle \\ \hat{\Omega}_S \vdash_{\hat{\Psi}_S;\hat{\Phi}_S} \hat{e}_{\text{parse}} \rightsquigarrow e_{\text{parse}} : \texttt{parr(Body;ParseResult(PPrPat))} \\ [\omega]e_{\text{parse}} \Downarrow e'_{\text{parse}} \qquad \hat{\Omega} \vdash^{\omega:\hat{\Omega}_S;\hat{\Psi}_S;\langle \mathcal{A}_S \uplus \hat{a} \hookrightarrow \texttt{defref}[a];\Phi_S, a \hookrightarrow \texttt{pptsm}(\rho;e_{\text{parse}})\rangle}_{\hat{\Psi};\hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma\end{array}}{\hat{\Omega} \vdash^{\Sigma}_{\hat{\Psi};\hat{\Phi}} \texttt{static syntax } \hat{a} \texttt{ at } \hat{\rho} \texttt{ for patterns by static } \hat{e}_{\text{parse}} \texttt{ in } \hat{M} \rightsquigarrow M : \sigma} \tag{6.1l}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\langle \mathcal{A};\Phi\rangle} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \rho \qquad \hat{\Omega} \vdash^{\Sigma}_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \epsilon;\Phi\rangle} \hat{M} \rightsquigarrow M : \sigma}{\hat{\Omega} \vdash^{\Sigma}_{\hat{\Psi};\langle \mathcal{A};\Phi\rangle} \texttt{standard let syntax } \hat{a} = \hat{e} \texttt{ for patterns in } \hat{M} \rightsquigarrow M : \sigma} \tag{6.1m}$$

$$\frac{\begin{array}{c}\hat{\Omega} \vdash^{\mathsf{Pat}}_{\langle \mathcal{A};\Phi\rangle} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \rho \\ \Sigma = \omega : \hat{\Omega}_S; \hat{\Psi}_S; \hat{\Phi}_S \qquad \hat{\Phi}_S = \langle \mathcal{A}_S; \Phi_S \rangle \\ \hat{\Omega} \vdash^{\omega:\hat{\Omega}_S;\hat{\Psi}_S;\langle \mathcal{A}_S \uplus \hat{a} \hookrightarrow \epsilon;\Phi_S\rangle}_{\hat{\Psi};\hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma\end{array}}{\hat{\Omega} \vdash^{\Sigma}_{\hat{\Psi};\hat{\Phi}} \texttt{static let syntax } \hat{a} = \hat{e} \texttt{ for patterns in } \hat{M} \rightsquigarrow M : \sigma} \tag{6.1n}$$

6.4.3 Metatheory

The metatheorem having to do with unexpanded module expressions was the Module Expansion theorem, Theorem C.44. This theorem continues to hold in miniVerse$_{\mathbf{PH}}$.

Theorem 6.1 (Module Expansion). *If* $\langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle \vdash^{\Sigma}_{\hat{\Psi};\hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma$ *then* $\Omega \vdash M : \sigma$.

Proof. By rule induction over Rules (6.1). In the following, let $\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle$.

Case (6.1a).

(1) $\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma'$ by assumption

(2) $\hat{\Omega} \vdash \sigma' <: \sigma$ by assumption

(3) $\Omega \vdash M : \sigma'$ by IH on (1)

(4) $\Omega \vdash M : \sigma$ by Rule (C.4a) on (3) and (2)

Case (6.1b) **through** (6.1f)**.** In each of these cases, we apply the IH over each module expansion premise, Theorem C.43 over each expression expansion premise and Theorem C.35 over each construction expansion premise, then apply the corresponding signature matching rule in Rules (C.4) and weakening as needed.

Case (6.1g) **through** (6.1n)**.** In each of these cases, we apply the IH to the module expansion premise.

$\square$

The rest of the metatheory is identical to that of miniVerse$_{\mathbf{P}}$.

Chapter 7

TSM Implicits

When applying a TSM, library clients must explicitly prefix each literal form with a TSM name and, in some cases, several parameters. In situations where the client is repeatedly applying a TSM to small literal forms, this can itself be costly. For example, list literals are often small, so applying `$intlist` repeatedly can be distracting and syntactically costly.

To further lower the syntactic cost of using TSMs, so that it compares to the syntactic cost of using derived forms built primitively into a language, VerseML allows clients to designate, for any type, one expression TSM and one pattern TSM as that type's *designated TSMs* within a delimited scope. When VerseML's *local type inference* system encounters a generalized literal form not prefixed by a TSM name (an *unadorned literal form*), it implicitly applies the TSM designated at the type that the expression or pattern is being checked against. This chapter will introduce TSM implicits first by example in Sec. 7.1 and then formally in Sec. 7.2.

7.1 TSM Implicits By Example

7.1.1 Designation and Usage

On Lines 1-2 of Figure 7.1, the client has *designated* the expression TSM `$rx` for implicit application to *unadorned literal forms* being checked against type `rx`, like the unadorned literal form on Line 5.

Similarly, on Line 3 of Figure 7.1 the client has designated the pattern TSM `$rx` for implicit application to unadorned pattern literal forms matching values of type `rx`, like the pattern form on Line 8.

Type annotations on TSM designations are technically redundant – the definition of the designated TSM determines the designated type. Annotations are included in our examples for readability.

Expression and pattern TSMs need not be designated together, nor have the same name if they are. However, this is a common idiom, so for convenience, VerseML also provides a derived designation form that combines the two designations in Figure 7.1:

```
implicit syntax $rx at rx in (* ... *) end
```

```
1  implicit syntax
2    $rx at rx for expressions
3    $rx at rx for patterns
4  in
5    val ssn : rx = /\d\d\d-\d\d-\d\d\d\d/
6    fun name_from_example_rx(r : rx) : option(string) =>
7      match r with
8        /@name: %_/ => Some name
9      | _ => None
10 end
```

Figure 7.1: An example of simple TSM implicits in VerseML

7.1.2 Analytic and Synthetic Positions

During typed expansion of a subexpression, e', of an expresssion, e, we say that e' appears in *analytic position* if the type that e' must have is determined by the surrounding context and its position within e. For example, an expression appearing as a function argument is in analytic position because the function's type determines the argument's type. Similarly, an expression may appear in analytic position due to a *type ascription*, either directly on the expression, or on a binding, as on Line 5 above.

If the type that e' must be assigned is not determined by the surrounding context – i.e. e' must be examined to synthesize its type – we instead say that the expression appears in a *synthetic position*. For example, a top-level expression, or an expression being bound without a type ascription, appears in synthetic position.

An expression of unadorned literal form is valid only in analytic position, because its type must be known to be able to determine the designated TSM that will control its expansion. For example, typed expansion of the following expression will fail because an expression of unadorned literal form appears in synthetic position:

```
val ssn = /\d\d\d-\d\d-\d\d\d\d/ (* INVALID *)
```

Patterns can always be of unadorned literal form in VerseML, because the scrutinee of a match expression is always in synthetic position, and so the type of value that each pattern appearing within the match expression must match is always known.

7.2 Bidirectional miniVerse$_{\mathsf{S}}$

To formalize TSM implicits, we will now develop a reduced calculus called *Bidirectional* miniVerse$_{\mathsf{S}}$. The full definition of this calculus is given in Appendix D. We choose to base our calculus on the simpler miniVerse$_{\mathsf{S}}$ calculus, rather than miniVerse$_{\mathsf{P}}$, to communicate the essential character of TSM implicits. Section 7.3 briefly considers the small changes that would be necessary to incorporate the same mechanism into a bidirectionally typed variant of miniVerse$_{\mathsf{P}}$.

Sort			Stylized Form	Description
UTyp	$\hat{\tau}$	$::=$	$\cdots$	(as in $\mathsf{miniVerse_S}$)
UExp	$\hat{e}$	$::=$	$\cdots$	(as in $\mathsf{miniVerse_S}$)
			$\mathtt{implicit\ syntax}\ \hat{a}\ \mathtt{for\ expressions\ in}\ \hat{e}$	seTSM designation
			$\mathtt{implicit\ syntax}\ \hat{a}\ \mathtt{for\ patterns\ in}\ \hat{e}$	spTSM designation
			$/b/$	seTSM unadorned literal
URule	$\hat{r}$	$::=$	$\cdots$	(as in $\mathsf{miniVerse_S}$)
UPat	$\hat{p}$	$::=$	$\cdots$	(as in $\mathsf{miniVerse_S}$)
			$/b/$	spTSM unadorned literal

Figure 7.2: Syntax of unexpanded terms in Bidirectional $\mathsf{miniVerse_S}$

7.2.1 Expanded Language

The Bidirectional $\mathsf{miniVerse_S}$ expanded language (XL) is the same as the $\mathsf{miniVerse_S}$ XL, which was described in Sections 4.2.1 through 4.2.3.

7.2.2 Syntax of the Unexpanded Language

The syntax of the Bidirectional $\mathsf{miniVerse_S}$ unexpanded language (UL) extends the syntax of the $\mathsf{miniVerse_S}$ UL as shown in Figure 7.2.

As in $\mathsf{miniVerse_S}$, there is also a textual syntax for the UL, characterized by the following condition:

Condition D.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* $\mathsf{parseUPat}(b) = \hat{p}$*.*

7.2.3 Bidirectionally Typed Expansion

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

Judgement Form	Description
$\hat{\Delta} \vdash \hat{\tau} \leadsto \tau\ \mathsf{type}$	$\hat{\tau}$ has well-formed expansion τ
$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau$	$\hat{e}$ has expansion e synthesizing type τ
$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Leftarrow \tau$	$\hat{e}$ has expansion e when analyzed against type τ
$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \leadsto r \Leftarrow \tau \Mapsto \tau'$	$\hat{r}$ has expansion r and takes values of type τ to values of type τ' when $\tau's$ is provided for analysis
$\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \leadsto p : \tau \dashv\!\vert\ \hat{\Gamma}$	$\hat{p}$ has expansion p and type τ and generates hypotheses $\hat{\Gamma}$

Type Expansion

Unexpanded type formation contexts, $\hat{\Delta}$, were defined in Sec. 3.2.6. The *type expansion judgement*, $\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \; \mathsf{type}$, is inductively defined as in miniVerse$_{\mathsf{S}}$ by Rules (B.5).

Bidirectionally Typed Expression and Rule Expansion

In order to clearly define the semantics of TSM implicits, we must make a judgmental distinction between *type synthesis* and *type analysis*. In the former, the type is determined from the term, while in the latter, the type is presumed known. Type systems that make this distinction are called *bidirectional type systems* [100]. (Pierce characterizes the idea as folklore predating his paper.)

The *typed expression expansion judgements*, $\hat{\Delta} \; \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau$, for type synthesis, and $\hat{\Delta} \; \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau$, for type analysis, and the typed rule expansion judgement, $\hat{\Delta} \; \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \rightsquigarrow r \Leftarrow \tau \Mapsto \tau'$, are defined mutually inductively by Rules (D.1), Rules (D.2) and Rule (D.3), respectively. We will reproduce only certain "interesting" rules below – the appendix gives the complete set of rules.

Subsumption Type analysis subsumes type synthesis according to the following *rule of subsumption*:

$$\frac{\hat{\Delta} \; \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau}{\hat{\Delta} \; \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau} \tag{D.2a}$$

In other words, when a type can be synthesized for an unexpanded expression, that unexpanded expression can also be analyzed against that type, producing the same expansion.

Type Ascription A *type ascription* can be placed on an unexpanded expression to specify the type that it should be analyzed against.

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \; \mathsf{type} \qquad \hat{\Delta} \; \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Delta} \; \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} : \hat{\tau} \rightsquigarrow e \Rightarrow \tau} \tag{D.1b}$$

Variables *Unexpanded typing contexts*, $\hat{\Gamma}$, were defined in Sec. 3.2.6. Identifiers that appear in $\hat{\Gamma}$ have the expansion and synthesize the type that $\hat{\Gamma}$ assigns to them.

$$\frac{}{\hat{\Delta} \; \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{x} \rightsquigarrow x \Rightarrow \tau} \tag{D.1a}$$

Value Binding We define let-binding of a value in synthetic or analytic position primitively in Bidirectional miniVerse$_{\mathsf{S}}$. The following rules govern this construct.

$$\frac{\hat{\Delta} \; \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \hat{\Delta} \; \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \rightsquigarrow e' \Rightarrow \tau'}{\hat{\Delta} \; \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{let val } \hat{x} = \hat{e} \texttt{ in } \hat{e}' \rightsquigarrow \texttt{ap(lam\{}\tau\texttt{\}(}x.e'\texttt{);}e\texttt{)} \Rightarrow \tau'} \tag{D.1c}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau \qquad \hat{\Delta}\ \hat{\Gamma}, \hat{x} \leadsto x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \leadsto e' \Leftarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{let}\,\texttt{val}\ \hat{x} = \hat{e}\ \texttt{in}\ \hat{e}' \leadsto \texttt{ap(lam\{}\tau\texttt{\}(}x.e'\texttt{);}e\texttt{)} \Leftarrow \tau'} \tag{D.2b}$$

Functions Functions with an argument type annotation can appear in synthetic position.

$$\frac{\hat{\Delta} \vdash \hat{\tau}_1 \leadsto \tau_1\ \mathsf{type} \qquad \hat{\Delta}\ \hat{\Gamma}, \hat{x} \leadsto x : \tau_1 \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau_2}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \lambda\hat{x}{:}\hat{\tau}_1.\hat{e} \leadsto \texttt{lam\{}\tau_1\texttt{\}(}x.e\texttt{)} \Rightarrow \texttt{parr(}\tau_1\texttt{;}\tau_2\texttt{)}} \tag{D.1d}$$

(In addition to such "half annotated" functions [27], it would be straightforward to include unannotated functions, $\lambda\hat{x}.\hat{e}$, which can appear only in analytic position. We leave these out for simplicity.)

Function applications can appear in synthetic position. The argument is analyzed against the argument type synthesized by the function.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_1 \leadsto e_1 \Rightarrow \texttt{parr(}\tau_2\texttt{;}\tau\texttt{)} \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_2 \leadsto e_2 \Leftarrow \tau_2}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_1(\hat{e}_2) \leadsto \texttt{ap(}e_1\texttt{;}e_2\texttt{)} \Rightarrow \tau} \tag{D.1e}$$

Pattern Matching The following rule governs match expressions, which must appear in analytic position.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau \qquad \{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \leadsto r_i \Leftarrow \tau \Mapsto \tau'\}_{1\leq i\leq n}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{match}\ \hat{e}\ \{\hat{r}_i\}_{1\leq i\leq n} \leadsto \texttt{match}[n]\texttt{(}e\texttt{;}\{r_i\}_{1\leq i\leq n}\texttt{)} \Leftarrow \tau'} \tag{D.2g}$$

The typed rule expansion judgement is defined by the following rule:

$$\frac{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \leadsto p : \tau \dashv\!\!\mid \langle\mathcal{G}';\Gamma'\rangle \qquad \langle\mathcal{D};\Delta\rangle\ \langle\mathcal{G} \uplus \mathcal{G}';\Gamma\cup\Gamma'\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Leftarrow \tau'}{\langle\mathcal{D};\Delta\rangle\ \langle\mathcal{G};\Gamma\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{p} \Rightarrow \hat{e} \leadsto \texttt{rule(}p.e\texttt{)} \Leftarrow \tau \Mapsto \tau'} \tag{D.3}$$

(In this simple calculus, it would also be possible to allow match expressions to appear in synthetic position – all of the branches would need to synthesize the same type. In a language with richer notions of type equality and subtyping, this requires greater care. To avoid this orthogonal concern, we do not formally consider this case.)

The pattern expansion judgement, $\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \leadsto p : \tau \dashv\!\!\mid \hat{\Gamma}$, is inductively defined by Rules (D.4), and operates as described in Chapter 4. There is one new rule, governing the newly introduced unadorned pattern literal form. We will return to this rule below.

Other Shared Forms Other constructs of shared form have similar bidirectional rules, given in the appendix.

TSMs seTSM contexts, $\hat{\Psi}$, take the form

$$\langle\mathcal{A};\Psi;\mathcal{I}\rangle$$

and spTSM contexts, $\hat{\Phi}$, take the form

$$\langle \mathcal{A}; \Phi; \mathcal{I} \rangle$$

where TSM identifier expansion contexts, $\mathcal{A}$, seTSM definition contexts, Ψ, and spTSM definition contexts, Φ, are defined as in miniVerse$_\mathsf{S}$. *TSM implicit designation contexts*, $\mathcal{I}$, are new to Bidirectional miniVerse$_\mathsf{S}$ and defined below.

Before considering TSM implicits, let us briefly review the rules for defining and explicitly applying TSMs. These rules are nearly identical to their counterparts in miniVerse$_\mathsf{S}$, differing only in that they have been made bidirectional.

TSMs can be defined in synthetic or analytic position. The rules for seTSMs are reproduced below (the rules for spTSMs are analagous – see appendix.)

$$\frac{\begin{array}{cc}\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \ \mathsf{type} & \emptyset\,\emptyset \vdash e_{\text{parse}} : \mathtt{parr(Body;ParseResultSE)} \\ e_{\text{parse}} \Downarrow e'_{\text{parse}} & \hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{setsm}(\tau; e'_{\text{parse}}); \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau'\end{array}}{\hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{for\ expressions}\ \{e_{\text{parse}}\}\ \mathtt{in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \tag{D.1k}$$

$$\frac{\begin{array}{cc}\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \ \mathsf{type} & \emptyset\,\emptyset \vdash e_{\text{parse}} : \mathtt{parr(Body;ParseResultSE)} \\ e_{\text{parse}} \Downarrow e'_{\text{parse}} & \hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{setsm}(\tau; e'_{\text{parse}}); \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'\end{array}}{\hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{for\ expressions}\ \{e_{\text{parse}}\}\ \mathtt{in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{D.2h}$$

The rule for explicitly applying an seTSM is reproduced below:

$$\frac{\begin{array}{c}\hat{\Psi} = \hat{\Psi}', \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{setsm}(\tau; e_{\text{parse}}) \\ b \downarrow_{\mathsf{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \mathtt{SuccessE} \cdot e_{\text{proto}} \qquad e_{\text{proto}} \uparrow_{\mathsf{PrExpr}} \grave{e} \\ \mathsf{seg}(\grave{e})\ \mathsf{segments}\ b \qquad \emptyset\,\emptyset \vdash^{\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; \hat{\Phi}; b} \grave{e} \rightsquigarrow e \Leftarrow \tau\end{array}}{\hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{a}\ \text{`}b\text{`} \rightsquigarrow e \Rightarrow \tau} \tag{D.1l}$$

Similarly, the rule for explicitly applying an spTSM is reproduced below:

$$\frac{\begin{array}{c}\hat{\Phi} = \hat{\Phi}', \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{sptsm}(\tau; e_{\text{parse}}) \\ b \downarrow_{\mathsf{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \mathtt{SuccessP} \cdot e_{\text{proto}} \qquad e_{\text{proto}} \uparrow_{\mathsf{PrPat}} \grave{p} \\ \mathsf{seg}(\grave{p})\ \mathsf{segments}\ b \qquad \grave{p} \rightsquigarrow p : \tau \dashv\!|^{\hat{\Delta}; \hat{\Phi}; b} \hat{\Gamma}\end{array}}{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{a}\ \text{`}b\text{`} \rightsquigarrow p : \tau \dashv\!| \hat{\Gamma}} \tag{D.4f}$$

TSM Implicits *TSM implicit designation contexts*, $\mathcal{I}$, are finite functions that map each type $\tau \in \mathsf{dom}(\mathcal{I})$ to the *TSM designation* $\tau \hookrightarrow a$, for some TSM name a. We write $\mathcal{I} \uplus \tau \hookrightarrow a$ for the TSM designation context that maps τ to $\tau \hookrightarrow a$ and defers to $\mathcal{I}$ for all other types (i.e. the previous designation, if any, is updated).

The following rules governs seTSM designation in synthetic and analytic position, respectively:

$$\frac{\hat{\Psi} = \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}}); \mathcal{I} \rangle \quad \hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle; \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{implicit syntax}\ \hat{a}\ \texttt{for expressions in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \quad \text{(D.1m)}$$

$$\frac{\hat{\Psi} = \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}}); \mathcal{I} \rangle \quad \hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{implicit syntax}\ \hat{a}\ \texttt{for expressions in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau'} \quad \text{(D.2i)}$$

Similarly, the following rules govern spTSM designation in synthetic and analytic position, respectively:

$$\frac{\hat{\Phi} = \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}}); \mathcal{I} \rangle \quad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle} \hat{e} \rightsquigarrow e \Rightarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{implicit syntax}\ \hat{a}\ \texttt{for patterns in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \quad \text{(D.1o)}$$

$$\frac{\hat{\Phi} = \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}}); \mathcal{I} \rangle \quad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{implicit syntax}\ \hat{a}\ \texttt{for patterns in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau'} \quad \text{(D.2l)}$$

The following rule determines the TSM designated at the type that the expression of unadorned literal form is being analyzed against and applies it implicitly:

$$\frac{\begin{array}{c}\hat{\Psi} = \langle \mathcal{A}; \Psi, a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle \\ b \downarrow_{\mathsf{Body}} e_{\text{body}} \quad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{SuccessE} \cdot e_{\text{proto}} \quad e_{\text{proto}} \uparrow_{\mathsf{PrExpr}} \grave{e} \\ \mathsf{seg}(\grave{e})\ \mathsf{segments}\ b \quad \emptyset\ \emptyset \vdash^{\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; \hat{\Phi}; b} \grave{e} \rightsquigarrow e \Leftarrow \tau\end{array}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} /b/ \rightsquigarrow e \Leftarrow \tau} \quad \text{(D.2j)}$$

Similarly, the following rule determines the TSM designated at the type that the pattern of unadorned literal form is matching against and applies it implicitly:

$$\frac{\begin{array}{c}\hat{\Phi} = \langle \mathcal{A}; \Phi, a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}}); \mathcal{I}, \tau \hookrightarrow a \rangle \\ b \downarrow_{\mathsf{Body}} e_{\text{body}} \quad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{SuccessP} \cdot e_{\text{proto}} \quad e_{\text{proto}} \uparrow_{\mathsf{PrPat}} \grave{p} \\ \mathsf{seg}(\grave{p})\ \mathsf{segments}\ b \quad \grave{p} \rightsquigarrow p : \tau \dashv\!|^{\hat{\Delta}; \hat{\Phi}; b}\ \hat{\Gamma}\end{array}}{\hat{\Delta} \vdash_{\hat{\Phi}} /b/ \rightsquigarrow p : \tau \dashv\!|\ \hat{\Gamma}} \quad \text{(D.4g)}$$

7.2.4 Bidirectional Proto-Expansion Validation

The syntax of proto-expansions was defined in Sec. 4.2.8.

The *bidirectional proto-expansion validation judgements* validate proto-terms and simultaneously generate their final expansions.

Judgement Form	Description
$\Delta \vdash^{\mathbb{T}} \grave{\tau} \rightsquigarrow \tau\ \mathsf{type}$	$\grave{\tau}$ has well-formed expansion τ
$\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau$	$\grave{e}$ has expansion e synthesizing type τ
$\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau$	$\grave{e}$ has expansion e when analyzed against type τ
$\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r} \rightsquigarrow r \Leftarrow \tau \Mapsto \tau'$	$\grave{r}$ has expansion r taking values of type τ to values of type τ'
$\grave{p} \rightsquigarrow p : \tau \dashv\vert^{\mathbb{P}} \hat{\Gamma}$	$\grave{p}$ has expansion p matching against τ generating assumptions $\hat{\Gamma}$

These judgements are defined by rules given in Appendix D.3.2. Most rules follow the corresponding typed expansion rule. The main rule of interest here is the rule governing references to spliced expressions, reproduced below:

$$\frac{\begin{array}{c}\emptyset \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \mathbb{E} = \langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle;\ \langle \mathcal{G}; \Gamma_{\mathsf{app}} \rangle;\ \hat{\Psi};\ \hat{\Phi};\ b \\ \mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e} \qquad \langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle\ \langle \mathcal{G}; \Gamma_{\mathsf{app}} \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau \\ \Delta \cap \Delta_{\mathsf{app}} = \emptyset \qquad \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma_{\mathsf{app}}) = \emptyset\end{array}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{splicede}[m; n; \grave{\tau}] \rightsquigarrow e \Rightarrow \tau} \tag{D.5k}$$

This rule is similar to Rule (B.10n), which governed references to spliced expressions in $\mathsf{miniVerse}_{\mathsf{S}}$. Notice that here, the unexpanded expression $\hat{e}$ is analyzed against the type τ.

7.2.5 Metatheory

Bidirectional $\mathsf{miniVerse}_{\mathsf{S}}$ enjoys metatheoretic properties analagous to those established for $\mathsf{miniVerse}_{\mathsf{S}}$. We state these properties below – the proofs are given in Appendix D.4.

The following theorem establishes that typed pattern expansion produces an expanded pattern that matches values of the specified type and generates the same hypotheses. It must be stated mutually with the corresponding theorem about proto-patterns, because the judgements are mutually defined.

Theorem D.9 (Typed Pattern Expansion)**.**

1. *If* $\langle \mathcal{D}; \Delta \rangle \vdash_{\langle \mathcal{A}; \Phi; \mathcal{I} \rangle} \hat{p} \rightsquigarrow p : \tau \dashv\vert \langle \mathcal{G}; \Gamma \rangle$ *then* $\Delta \vdash p : \tau \dashv\vert \Gamma$.
2. *If* $\grave{p} \rightsquigarrow p : \tau \dashv\vert^{\langle \mathcal{D}; \Delta \rangle; \langle \mathcal{A}; \Phi \rangle; b} \langle \mathcal{G}; \Gamma \rangle$ *then* $\Delta \vdash p : \tau \dashv\vert \Gamma$.

The following theorem establishes that bidirectionally typed expression and rule expansion produces expanded expressions and rules of the appropriate type under the appropriate contexts. These statements must be stated mutually with the corresponding statements about birectional proto-expression and proto-rule validation because the judgements are mutually defined.

Theorem D.10 (Typed Expression and Rule Expansion)**.**

1. (a) *If* $\langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau$ *then* $\Delta\ \Gamma \vdash e : \tau$.
   (b) *If* $\langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau$ *and* $\Delta \vdash \tau\ \mathsf{type}$ *then* $\Delta\ \Gamma \vdash e : \tau$.
   (c) *If* $\langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{r} \rightsquigarrow r \Leftarrow \tau \Mapsto \tau'$ *and* $\Delta \vdash \tau'\ \mathsf{type}$ *then* $\Delta\ \Gamma \vdash r : \tau \Mapsto \tau'$.
2. (a) *If* $\Delta\ \Gamma \vdash^{\langle \mathcal{D}; \Delta_{app} \rangle; \langle \mathcal{G}; \Gamma_{app} \rangle; \hat{\Psi}; \hat{\Phi}; b} \grave{e} \rightsquigarrow e \Rightarrow \tau$ *and* $\Delta \cap \Delta_{app} = \emptyset$ *and* $dom(\Gamma) \cap dom(\Gamma_{app}) = \emptyset$ *then* $\Delta \cup \Delta_{app}\ \Gamma \cup \Gamma_{app} \vdash e : \tau$.
   (b) *If* $\Delta\ \Gamma \vdash^{\langle \mathcal{D}; \Delta_{app} \rangle; \langle \mathcal{G}; \Gamma_{app} \rangle; \hat{\Psi}; \hat{\Phi}; b} \grave{e} \rightsquigarrow e \Leftarrow \tau$ *and* $\Delta \vdash \tau\ \mathsf{type}$ *and* $\Delta \cap \Delta_{app} = \emptyset$ *and* $dom(\Gamma) \cap dom(\Gamma_{app}) = \emptyset$ *then* $\Delta \cup \Delta_{app}\ \Gamma \cup \Gamma_{app} \vdash e : \tau$.

(c) *If* $\Delta\ \Gamma \vdash^{\langle \mathcal{D};\Delta_{app}\rangle;\langle \mathcal{G};\Gamma_{app}\rangle;\hat{\Psi};\hat{\Phi};b} \grave{r} \rightsquigarrow r \Leftarrow \tau \Mapsto \tau'$ *and* $\Delta \vdash \tau'$ type *and* $\Delta \cap \Delta_{app} = \emptyset$ *and* $dom(\Gamma) \cap dom(\Gamma_{app}) = \emptyset$ *then* $\Delta \cup \Delta_{app}\ \Gamma \cup \Gamma_{app} \vdash r : \tau \Mapsto \tau'$.

The following theorem establishes abstract reasoning principles for implicitly applied expression TSMs. These are analagous to those described in Section 3.2.11 for explicitly applied expression TSMs.

Theorem D.13 (seTSM Abstract Reasoning Principles - Implicit Application)**.** *If*

$$\langle \mathcal{D};\Delta\rangle\ \langle \mathcal{G};\Gamma\rangle \vdash_{\hat{\Psi};\hat{\Phi}} /b/ \rightsquigarrow e \Leftarrow \tau$$

then:

1. ***(Typing 1)*** $\hat{\Psi} = \langle \mathcal{A};\Psi, a \hookrightarrow \texttt{setsm}(\tau; e_{parse});\mathcal{I} \uplus \tau \hookrightarrow a\rangle$ *and* $\Delta\ \Gamma \vdash e : \tau$
2. $b \downarrow_{\mathsf{Body}} e_{body}$
3. $e_{parse}(e_{body}) \Downarrow \texttt{inj}[\texttt{SuccessE}](e_{proto})$
4. $e_{proto} \uparrow_{\mathsf{PrExpr}} \grave{e}$
5. ***(Segmentation)*** $\mathsf{seg}(\grave{e})$ segments b
6. $\mathsf{summary}(\grave{e}) = \{\texttt{splicedt}[m'_i; n'_i]\}_{0\leq i<n_{ty}} \cup \{\texttt{splicede}[m_i; n_i; \grave{\tau}_i]\}_{0\leq i<n_{exp}}$
7. ***(Typing 2)*** $\{\langle \mathcal{D};\Delta\rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow \tau'_i \text{ type}\}_{0\leq i<n_{ty}}$ *and* $\{\Delta \vdash \tau'_i \text{ type}\}_{0\leq i<n_{ty}}$
8. ***(Typing 3)*** $\{\emptyset \vdash^{\langle \mathcal{D};\Delta\rangle;b} \grave{\tau}_i \rightsquigarrow \tau_i \text{ type}\}_{0\leq i<n_{exp}}$ *and* $\{\Delta \vdash \tau_i \text{ type}\}_{0\leq i<n_{exp}}$
9. ***(Typing 4)*** $\{\langle \mathcal{D};\Delta\rangle\ \langle \mathcal{G};\Gamma\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \mathsf{parseUExp}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow e_i \Leftarrow \tau_i\}_{0\leq i<n_{exp}}$ *and* $\{\Delta\ \Gamma \vdash e_i : \tau_i\}_{0\leq i<n_{exp}}$
10. ***(Capture Avoidance)*** $e = [\{\tau'_i/t_i\}_{0\leq i<n_{ty}}, \{e_i/x_i\}_{0\leq i<n_{exp}}]e'$ *for some* $\{t_i\}_{0\leq i<n_{ty}}$ *and* $\{x_i\}_{0\leq i<n_{exp}}$ *and* e'
11. ***(Context Independence)*** $\mathsf{fv}(e') \subset \{t_i\}_{0\leq i<n_{ty}} \cup \{x_i\}_{0\leq i<n_{exp}}$

Similarly, the following theorem establishes abstract reasoning principles for implicitly applied pattern TSMs. These are analagous to those described in Sec. 4.2.10 for explicitly applied pattern TSMs.

Theorem D.16 (spTSM Abstract Reasoning Principles - Implicit Application)**.** *If*

$$\hat{\Delta} \vdash_{\hat{\Phi}} /b/ \rightsquigarrow p : \tau \dashv\!\mid \hat{\Gamma}$$

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)*** $\hat{\Phi} = \langle \mathcal{A};\Phi, a \hookrightarrow \texttt{sptsm}(\tau; e_{parse});\mathcal{I}, \tau \hookrightarrow a\rangle$ *and* $\Delta \vdash p : \tau \dashv\!\mid \Gamma$
2. $b \downarrow_{\mathsf{Body}} e_{body}$
3. $e_{parse}(e_{body}) \Downarrow \texttt{inj}[\texttt{SuccessP}](e_{proto})$
4. $e_{proto} \uparrow_{\mathsf{PrPat}} \grave{p}$
5. ***(Segmentation)*** $\mathsf{seg}(\grave{p})$ segments b
6. $\mathsf{summary}(\grave{p}) = \{\texttt{splicedt}[n'_i; m'_i]\}_{0\leq i<n_{ty}} \cup \{\texttt{splicedp}[m_i; n_i; \grave{\tau}_i]\}_{0\leq i<n_{pat}}$
7. ***(Typing 2)*** $\{\hat{\Delta} \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow \tau'_i \text{ type}\}_{0\leq i<n_{ty}}$ *and* $\{\Delta \vdash \tau'_i \text{ type}\}_{0\leq i<n_{ty}}$
8. ***(Typing 3)*** $\{\emptyset \vdash^{\hat{\Delta};b} \grave{\tau}_i \rightsquigarrow \tau_i \text{ type}\}_{0\leq i<n_{pat}}$ *and* $\{\Delta \vdash \tau_i \text{ type}\}_{0\leq i<n_{pat}}$
9. ***(Typing 4)*** $\{\hat{\Delta} \vdash_{\hat{\Phi}} \mathsf{parseUPat}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow p_i : \tau_i \dashv\!\mid \hat{\Gamma}_i\}_{0\leq i<n_{pat}}$
10. ***(No Hidden Bindings)*** $\hat{\Gamma} = \biguplus_{0\leq i<n_{pat}} \hat{\Gamma}_i$

7.3 Parametric TSM Implicits

Incorporating simple implicits into a bidirectionally typed dialect of $\mathsf{miniVerse}_{\mathbf{P}}$ would require that the implicit context, $\mathcal{I}$, be a finite function from equivalence classes of types to TSM expressions, ϵ (rather than from syntactic types, τ, to TSM names, a.)

We consider a more sophisticated mechanism that allows a TSM implicit designation itself to operate over a parameterized family of types as future work in Sec. 8.2.5.

Chapter 8

Discussion & Future Directions

8.1 Summary

In summary, TSMs allow library providers to programmatically control the expansion of generalized expression and pattern literal forms. A proto-expansion validation step allows library clients to reason about types and binding without examining the generated expansions in complete detail. Instead, the client needs only to be made aware of the type annotation on the applied TSM and the *splice summary* of the generated proto-expansion, which locates and assigns a type to each spliced term within the literal body. This information can be communicated to the client using secondary notation (e.g. colors, as in this document) together with a simple service for querying the type of an expression similar to that available in many program editors today.

We developed several core calculi in order to formally characterize the mechanisms of TSM definition and application. In particular, Chapters 3 and 4 developed $\mathsf{miniVerse}_{\mathbf{S}}$, which communicates the central concepts of typed expansion and proto-expansion validation for expression and pattern TSMs respectively. Chapter 5 then introduced $\mathsf{miniVerse}_{\mathbf{P}}$, which added type and module parameters. Chapter 6 introduced $\mathsf{miniVerse}_{\mathbf{PH}}$, which adds the notion of a static environment shared between TSM definitions. This makes the job of the TSM provider easier, by giving them access to libraries (including those that themselves export TSM definitions.) Finally, Chapter 7 introduced Bidirectional $\mathsf{miniVerse}_{\mathbf{S}}$, which supports a mechanism of TSM implicits that further decreases the syntactic cost of applying a TSM.

8.2 Future Directions

8.2.1 Integration Into a Full-Scale Language Definition

We left many orthogonal language features out of our calculi, for the sake of simplicity and to focus our exposition on our novel contributions. We leave the work of integrating TSMs into a full-scale language definition to the future.

8.2.2 Constraint-Based Type Inference

ML uses a constraint-based type inference system *a la* Damas-Milner [32]. We conjecture that the mechanisms described up through Chapter 6 are compatible with such a system – most simply, by generating the constraints from the final expansion. Actually, it should be possible to perform type inference abstractly, using only the information in the splice summary. We leave the evaluation of this conjecture as future work.

The mechanism of TSM implicits developed in Chapter 7 assumed a bidirectional type system, i.e. one that only locally infers types [100]. It may be possible to integrate implicit TSM dispatch with a constraint-based type inference system, but the approach to take is less clear. We leave the exploration of this question as future work.

8.2.3 Integration Into Languages From Other Design Traditions

We conjecture that all of the mechanisms we have described could be integrated into dependently typed functional languages, e.g. Coq [86], but leave the necessary technical developments as future work.

The mechanisms described in Chapter 3, Chapter 5 and Chapter 6 could also be adapted for use in languages from the imperative and object-oriented traditions without difficulty. Similarly, these constructs could be adapted for use in dynamic languages like Racket as well by eliminating the type annotation (thereby treating the language as "uni-typed" [61, 107].) In such a language (and, in fact, even in a language with richer type structure), it would be useful to be able to annotate a TSM with a dynamic contract governing all generated expansions [47].

The mechanism of TSM implicits introduced in Chapter 7 assumes a bidirectionally typed unexpanded language. A number of full-scale languages are bidirectionally typed, notably including Scala [93]. However, Scala supports subtyping. Subtyping would complicate the question of implicit TSM dispatch, because there may be designations at multiple supertypes of a given type.

Various forms of *ad hoc* polymorphism, e.g. function/method overloading or type classes [58], would also complicate the question of implicit TSM dispatch. One approach that may be worth exploring in future work would associate implicits with a type class, rather than with an individual type.

8.2.4 Module Expression Syntax Macros

We did not consider situations where a library clients wants to define syntactic sugar for module expressions matching a given signature. It should be possible to "replicate" the mechanisms developed up through Chapter 6 at the level of module expressions without difficulty. Implicit dispatch would be problematic at the module level, again because signatures are related by a notion of subtyping.

8.2.5 Parameterized Implicit Designations

The mechanism of TSM implicits developed in Chapter 7 allows the client to designate a TSM at a single type. A more sophisticated mechanism would allow for parameterization of the TSM implicit designation itself, so that it could operate across a parameterized family of types. For example, we may want to be able to implicitly apply the parametric TSM `$list'` at *all* list types, with the parameters determined implicitly from the type supplied for analysis.

Naïvely, we might imagine a designation that quantifies over modules, `L`, and types, `'a`, like this:

```
implicit syntax (L : LIST) 'a => $list' L 'a in (* ... *) end
```

When encountering an unadorned literal form, the implicit dispatch mechanism must instantiate each implicit parameter from the type provided for analysis. The problem is that there does not always exist a unique such instantiation. For example, the type expression `list(int)` makes no reference to any module matching the `LIST` signature, and there may be many such modules in context, so we cannot uniquely instantiate `L`.

To solve this problem, we would need to define a unique *normal form* that serves as a representative for each equivalence class of types. A designation is deemed invalid if the normal form of its type does not mention every designation parameter. For example, the normal form of `L.list('a)` does not mention `L` (because `LIST.list` is not abstract), so the implicit designation above can simply be deemed invalid.

The following designation does not require instantiating a module variable, so it would be valid under this restriction (recalling that `$list` was defined as a synonym for `$list' List`):

```
implicit syntax 'a => $list 'a in (* ... *) end
```

The normal form of an abstract type would necessarily mention a module path, so the following parametric designation would also be valid:

```
implicit syntax (R : RX) => $r R in (* ... *) end
```

It may be possible to use Crary's method for representing terms of the dependent singleton calculus in a beta-normal, eta-long form for this purpose [30]. Defining pattern matching over types of normal form, and incorporating this mechanism into the implicit dispatch mechanism, is left as future work.

8.2.6 Exportable Implicit Designations

Implicit designations cannot be exported from a library, because different libraries might define conflicting designations. This can be inconvenient for clients.

This restriction is perhaps too severe in cases where the designation is at a type generated from within the same library. In such situation, it would be safe to export an implicit designation because no other library could do the same. We have explored this approach in a simple calculus that supports nominal type generativity [96]. We say that the syntax macro associated with a nominal type defines that type's *type-specific* *language* (TSL). We leave as future work the question of adapting this mechanism to support generative type abstraction as described in Chapter 5.

8.2.7 Mechanically Reasoning About Parse Functions

A correct parse function never returns an encoding of a proto-expansion that fails proto-expansion validation. This invariant cannot be enforced by the simple type systems we have considered in this document. Using a proof assistant, it would be possible to verify that a parse function generates only encodings of valid proto-expansions. Alternatively, in a dependently typed setting, the type of the parse function itself could be enriched so as to enforce this invariant intrinsically. We leave the details of this approach as future work.

A related problem is that a parse function might diverge. Again, we can either prove that the parse function does not diverge extrinsically, or define the parse function using a total language.

Parse functions implement some intended syntax definition. It would be useful to be able to state the syntax definition separately as a formal structure, and then prove that the parse function implements it correctly. In fact, in most cases, it should be possible to generate the parse function directly from the syntax definition using a parser generator. In this case, it would be useful to be able to mechanically prove the parser generator correct.

It would also be useful to develop the notion of an *splice summary specification*, i.e. a specification of the splice summary that should result from a well-formed string. This could be combined with a grammar like the one shown in Figure 6.3, with the non-terminals representing spliced terms annotated with types.

8.2.8 Refactoring Unexpanded Terms

A crucial distinction is between identifiers, which appear in unexpanded terms, and variables, which appear in expanded terms. Variables are given meaning by substitution, and ABTs are identified only up to renaming of bound variables. In contrast, identifiers are given meaning only by expansion to variables and there is no notion of renaming or substitution.

It would be useful to support an identifier renaming operation for the purposes of automatic refactoring [88]. The simplest solution would be to use the splice summaries to locate spliced terms, and then perform the renaming directly within the literal body. The problem is that there is no guarantee that the parse function will produce an alpha-equivalent expansion after such a renaming operation has been performed. Similar concerns about invariance come up for other kinds of refactorings.

There are three approaches one might take to avoid this problem. The simplest approach is for the renaming operation to re-run the parse function and check that the expansion it generates is related to the previously generated expansion as expected. Alternatively, we might seek to mechanically verify that the parse function is invariant to refactoring, either intrinsically or extrinsically as discussed in Sec. 8.2.7. A third approach would be to require that the TSM be defined using a grammar formalism that precludes inspection of the form of spliced expressions by construction. Exploration of these approaches is a promising avenue for future work.

8.2.9 Integration with Editor Services

Program editors often seek to provide feedback to the programmer about the syntax and semantics of the program being written. Questions remain about how various editor services should interact with TSMs. In the examples in this document, we colored spliced terms black and all other segments of a literal body some other uniform color (e.g. green.) A more sophisticated approach would allow the TSM to define its own syntax highlighting logic governing these non-spliced segments.

Another concern has to do with performance: naïvely, a program editor would need to re-run the corresponding parse function on each edit that modified a literal body. Ideally, it should be possible to incrementally compute the resulting change to the splice summary as a function of the change to the literal body [54].

8.2.10 Controlled Shadowing

The prohibition on capture makes it impossible to define new binding constructs. For example, consider Haskell's **do**-notation for monadic structures:

```
do x1 <- action1
   x2 <- action2
   action3 x1 x2
```

This desugars to:

```
action1 >>= \ x1 -> action2 >>= \ x2 -> action3 x1 x2
```

where `>>=` is infix application of the *bind* function and `\ x1 -> e` is Haskell's syntax for lambda abstraction.

If we naïvely attempted to define something like **do**-notation using a TSM, the prohibition on capture would prevent `x1` from being visible within `action2`.

Completely relaxing the prohibition on capture would be unreasonable. Instead, we conjecture that there are two important constraints that need to be enforced:

1. Identifiers that can be captured by spliced terms must themselves appear in the literal body. This leaves control over naming entirely to the client programmer.
2. The splice summary must state which of these are available to each spliced term.

We leave the technical details of this proposal as future work. We also leave open the question of how an editor should best convey the set of available identifiers within a spliced term to the programmer.

LaTeX Source Code and Updates

The LaTeX sources for this document can be found at the following URL:

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

The latest version of this document can be downloaded from the following URL:

https://github.com/cyrus-/thesis/raw/master/omar-thesis.pdf

Any errors or omissions can be reported using GitHub's issue tracker, or by sending an email to the author:

comar@cs.cmu.edu

Appendix

Appendix A

Conventions

A.1 Typographic Conventions

We adopt *PFPL*'s typographic conventions for operational forms throughout the paper [61]. In particular, the names of operators and indexed families of operators are written in `typewriter font`, indexed families of operators specify indices within [braces], and term arguments are grouped arbitrarily (roughly, by sort) using `{`curly braces`}` and `(`rounded braces`)`. We write $p.e$ for expressions binding the variables that appear in the pattern p.

We write $\{i \hookrightarrow \tau_i\}_{i \in L}$ for a sequence of arguments τ_i, one for each $i \in L$, and similarly for arguments of other valences. Operations that are parameterized by label sets, e.g. $\texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})$, are identified up to mutual reordering of the label set and the corresponding argument sequence. Similarly, we write $\{i \hookrightarrow J_i\}_{i \in L}$ for the finite set of derivations J_i for each $i \in L$.

We write $\{r_i\}_{1 \le i \le n}$ for sequences of $n \ge 0$ rule arguments, and similarly for other finite sequences.

Empty finite sets and finite functions are written $\emptyset$, or omitted entirely within judgements, and non-empty finite sets and finite functions are written as comma-separated sequences identified up to exchange and contraction.

Appendix B

miniVerse$_{\mathbf{SE}}$ and miniVerse$_{\mathbf{S}}$

This section defines miniVerse$_{\mathbf{S}}$, the language of Chapter 4. The language of Chapter 3, miniVerse$_{\mathbf{SE}}$, can be recovered by omitting the segments typeset in a gray backgrounds below.

B.1 Expanded Language (XL)

B.1.1 Syntax

Sort			Operational Form	Description
Typ	τ	$::=$	t	variable
			$\texttt{parr}(\tau;\tau)$	partial function
			$\texttt{all}(t.\tau)$	polymorphic
			$\texttt{rec}(t.\tau)$	recursive
			$\texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})$	labeled product
			$\texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i\in L})$	labeled sum
Exp	e	$::=$	x	variable
			$\texttt{lam}\{\tau\}(x.e)$	abstraction
			$\texttt{ap}(e;e)$	application
			$\texttt{tlam}(t.e)$	type abstraction
			$\texttt{tap}\{\tau\}(e)$	type application
			$\texttt{fold}(e)$	fold
			$\texttt{unfold}(e)$	unfold
			$\texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L})$	labeled tuple
			$\texttt{prj}[\ell](e)$	projection
			$\texttt{inj}[\ell](e)$	injection
			$\texttt{case}[L](e;\{i \hookrightarrow x_i.e_i\}_{i\in L})$	case analysis
			$\texttt{match}[n](e;\{r_i\}_{1\leq i\leq n})$	match
Rule	r	$::=$	$\texttt{rule}(p.e)$	rule
Pat	p	$::=$	x	variable pattern
			$\texttt{wildp}$	wildcard pattern
			$\texttt{foldp}(p)$	fold pattern
			$\texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i\in L})$	labeled tuple pattern
			$\texttt{injp}[\ell](p)$	injection pattern

B.1.2 Statics

Type formation contexts, Δ, are finite sets of hypotheses of the form t type. We write Δ, t type when t type $\notin \Delta$ for Δ extended with the hypothesis t type.

Typing contexts, Γ, are finite functions that map each variable $x \in \text{dom}(\Gamma)$, where $\text{dom}(\Gamma)$ is a finite set of variables, to the hypothesis $x : \tau$, for some τ. We write $\Gamma, x : \tau$, when $x \notin \text{dom}(\Gamma)$, for the extension of Γ with a mapping from x to $x : \tau$, and $\Gamma \cup \Gamma'$ when $\text{dom}(\Gamma) \cap \text{dom}(\Gamma') = \emptyset$ for the typing context mapping each $x \in \text{dom}(\Gamma) \cup \text{dom}(\Gamma')$ to $x : \tau$ if $x : \tau \in \Gamma$ or $x : \tau \in \Gamma'$. We write $\Delta \vdash \Gamma$ ctx if every type in Γ is well-formed relative to Δ.

Definition B.1 (Typing Context Formation). $\Delta \vdash \Gamma$ ctx *iff for each hypothesis* $x : \tau \in \Gamma$, *we have* $\Delta \vdash \tau$ type.

$\boxed{\Delta \vdash \tau \;\mathsf{type}}$ τ is a well-formed type

$$\frac{}{\Delta, t\;\mathsf{type} \vdash t\;\mathsf{type}} \tag{B.1a}$$

$$\frac{\Delta \vdash \tau_1\;\mathsf{type} \qquad \Delta \vdash \tau_2\;\mathsf{type}}{\Delta \vdash \mathtt{parr}(\tau_1;\tau_2)\;\mathsf{type}} \tag{B.1b}$$

$$\frac{\Delta, t\;\mathsf{type} \vdash \tau\;\mathsf{type}}{\Delta \vdash \mathtt{all}(t.\tau)\;\mathsf{type}} \tag{B.1c}$$

$$\frac{\Delta, t\;\mathsf{type} \vdash \tau\;\mathsf{type}}{\Delta \vdash \mathtt{rec}(t.\tau)\;\mathsf{type}} \tag{B.1d}$$

$$\frac{\{\Delta \vdash \tau_i\;\mathsf{type}\}_{i\in L}}{\Delta \vdash \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})\;\mathsf{type}} \tag{B.1e}$$

$$\frac{\{\Delta \vdash \tau_i\;\mathsf{type}\}_{i\in L}}{\Delta \vdash \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i\in L})\;\mathsf{type}} \tag{B.1f}$$

$\boxed{\Delta\;\Gamma \vdash e : \tau}$ e is assigned type τ

$$\frac{}{\Delta\;\Gamma, x : \tau \vdash x : \tau} \tag{B.2a}$$

$$\frac{\Delta \vdash \tau\;\mathsf{type} \qquad \Delta\;\Gamma, x : \tau \vdash e : \tau'}{\Delta\;\Gamma \vdash \mathtt{lam}\{\tau\}(x.e) : \mathtt{parr}(\tau;\tau')} \tag{B.2b}$$

$$\frac{\Delta\;\Gamma \vdash e_1 : \mathtt{parr}(\tau;\tau') \qquad \Delta\;\Gamma \vdash e_2 : \tau}{\Delta\;\Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau'} \tag{B.2c}$$

$$\frac{\Delta, t\;\mathsf{type}\;\Gamma \vdash e : \tau}{\Delta\;\Gamma \vdash \mathtt{tlam}(t.e) : \mathtt{all}(t.\tau)} \tag{B.2d}$$

$$\frac{\Delta\;\Gamma \vdash e : \mathtt{all}(t.\tau) \qquad \Delta \vdash \tau'\;\mathsf{type}}{\Delta\;\Gamma \vdash \mathtt{tap}\{\tau'\}(e) : [\tau'/t]\tau} \tag{B.2e}$$

$$\frac{\Delta\;\Gamma \vdash e : [\mathtt{rec}(t.\tau)/t]\tau}{\Delta\;\Gamma \vdash \mathtt{fold}(e) : \mathtt{rec}(t.\tau)} \tag{B.2f}$$

$$\frac{\Delta\;\Gamma \vdash e : \mathtt{rec}(t.\tau)}{\Delta\;\Gamma \vdash \mathtt{unfold}(e) : [\mathtt{rec}(t.\tau)/t]\tau} \tag{B.2g}$$

$$\frac{\{\Delta\;\Gamma \vdash e_i : \tau_i\}_{i\in L}}{\Delta\;\Gamma \vdash \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L}) : \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})} \tag{B.2h}$$

$$\frac{\Delta\,\Gamma \vdash e : \texttt{prod}[L,\ell](\{i \hookrightarrow \tau_i\}_{i\in L}; \ell \hookrightarrow \tau)}{\Delta\,\Gamma \vdash \texttt{prj}[\ell](e) : \tau} \tag{B.2i}$$

$$\frac{\Delta\,\Gamma \vdash e : \tau}{\Delta\,\Gamma \vdash \texttt{inj}[\ell](e) : \texttt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i\in L}; \ell \hookrightarrow \tau)} \tag{B.2j}$$

$$\frac{\Delta\,\Gamma \vdash e : \texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i\in L}) \qquad \{\Delta\,\Gamma, x_i : \tau_i \vdash e_i : \tau\}_{i\in L}}{\Delta\,\Gamma \vdash \texttt{case}[L](e; \{i \hookrightarrow x_i.e_i\}_{i\in L}) : \tau} \tag{B.2k}$$

$$\frac{\Delta\,\Gamma \vdash e : \tau \qquad \{\Delta\,\Gamma \vdash r_i : \tau \Mapsto \tau'\}_{1\le i\le n}}{\Delta\,\Gamma \vdash \texttt{match}[n](e; \{r_i\}_{1\le i\le n}) : \tau'} \tag{B.2l}$$

$\boxed{\Delta\,\Gamma \vdash r : \tau \Mapsto \tau'}$ r takes values of type τ to values of type τ'

$$\frac{\Delta \vdash p : \tau \dashv\!\mid \Gamma' \qquad \Delta\,\Gamma \cup \Gamma' \vdash e : \tau'}{\Delta\,\Gamma \vdash \texttt{rule}(p.e) : \tau \Mapsto \tau'} \tag{B.3}$$

Rule (B.3) is defined mutually inductively with Rules (B.2).

$\boxed{\Delta \vdash p : \tau \dashv\!\mid \Gamma}$ p matches values of type τ and generates hypotheses Γ

$$\frac{}{\Delta \vdash x : \tau \dashv\!\mid x : \tau} \tag{B.4a}$$

$$\frac{}{\Delta \vdash \texttt{wildp} : \tau \dashv\!\mid \emptyset} \tag{B.4b}$$

$$\frac{\Delta \vdash p : [\texttt{rec}(t.\tau)/t]\tau \dashv\!\mid \Gamma}{\Delta \vdash \texttt{foldp}(p) : \texttt{rec}(t.\tau) \dashv\!\mid \Gamma} \tag{B.4c}$$

$$\frac{\{\Delta \vdash p_i : \tau_i \dashv\!\mid \Gamma_i\}_{i\in L}}{\Delta \vdash \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i\in L}) : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L}) \dashv\!\mid \cup_{i\in L}\,\Gamma_i} \tag{B.4d}$$

$$\frac{\Delta \vdash p : \tau \dashv\!\mid \Gamma}{\Delta \vdash \texttt{injp}[\ell](p) : \texttt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i\in L}; \ell \hookrightarrow \tau) \dashv\!\mid \Gamma} \tag{B.4e}$$

Metatheory

The rules above are syntax-directed, so we assume an inversion lemma for each rule as needed without stating it separately or proving it explicitly. The following standard lemmas also hold.

The Weakening Lemma establishes that extending the context with unnecessary hypotheses preserves well-formedness and typing.

Lemma B.2 (Weakening)**.**

1. *If* $\Delta \vdash \tau$ type *then* Δ, t type $\vdash \tau$ type.
2. (a) *If* $\Delta\ \Gamma \vdash e : \tau$ *then* Δ, t type $\Gamma \vdash e : \tau$.
   (b) *If* $\Delta\ \Gamma \vdash r : \tau \Mapsto \tau'$ *then* Δ, t type $\Gamma \vdash r : \tau \Mapsto \tau'$.
3. (a) *If* $\Delta\ \Gamma \vdash e : \tau$ *and* $\Delta \vdash \tau''$ type *then* $\Delta\ \Gamma, x : \tau'' \vdash e : \tau$.
   (b) *If* $\Delta\ \Gamma \vdash r : \tau \Mapsto \tau'$ *and* $\Delta \vdash \tau''$ type *then* $\Delta\ \Gamma, x : \tau'' \vdash r : \tau \Mapsto \tau'$.
4. *If* $\Delta \vdash p : \tau \dashV \Gamma$ *then* $p : \tau \dashV \Gamma$.

Proof Sketch.

1. By rule induction over Rules (B.1).
2. By mutual rule induction over Rules (B.2) and Rule (B.3), and part 1.
3. By mutual rule induction over Rules (B.2) and Rule (B.3), and part 1.
4. By rule induction over Rules (B.4).

□

The pattern typing judgement is *linear* in the pattern typing context, i.e. it does *not* obey weakening of the pattern typing context. This is to ensure that the pattern typing context captures exactly those hypotheses generated by a pattern, and no others.

The Substitution Lemma establishes that substitution of a well-formed type for a type variable, or an expanded expression of the appropriate type for an expanded expression variable, preserves well-formedness and typing.

Lemma B.3 (Substitution)**.**

1. *If* Δ, t type $\vdash \tau$ type *and* $\Delta \vdash \tau'$ type *then* $\Delta \vdash [\tau'/t]\tau$ type.
2. (a) *If* Δ, t type $\Gamma \vdash e : \tau$ *and* $\Delta \vdash \tau'$ type *then* $\Delta\ [\tau'/t]\Gamma \vdash [\tau'/t]e : [\tau'/t]\tau$.
   (b) *If* Δ, t type $\Gamma \vdash r : \tau \Mapsto \tau''$ *and* $\Delta \vdash \tau'$ type *then* $\Delta\ [\tau'/t]\Gamma \vdash [\tau'/t]r : [\tau'/t]\tau \Mapsto [\tau'/t]\tau''$.
3. (a) *If* $\Delta\ \Gamma, x : \tau' \vdash e : \tau$ *and* $\Delta\ \Gamma \vdash e' : \tau'$ *then* $\Delta\ \Gamma \vdash [e'/x]e : \tau$.
   (b) *If* $\Delta\ \Gamma, x : \tau' \vdash r : \tau \Mapsto \tau''$ *and* $\Delta\ \Gamma \vdash e' : \tau''$ *then* $\Delta\ \Gamma \vdash [e'/x]r : \tau \Mapsto \tau''$.

Proof Sketch.

1. By rule induction over Rules (B.1).
2. By mutual rule induction over Rules (B.2) and Rule (B.3).
3. By mutual rule induction over Rules (B.2) and Rule (B.3).

□

The Decomposition Lemma is the converse of the Substitution Lemma.

Lemma B.4 (Decomposition)**.**

1. *If* $\Delta \vdash [\tau'/t]\tau$ type *and* $\Delta \vdash \tau'$ type *then* Δ, t type $\vdash \tau$ type.
2. (a) *If* $\Delta\ [\tau'/t]\Gamma \vdash [\tau'/t]e : [\tau'/t]\tau$ *and* $\Delta \vdash \tau'$ type *then* Δ, t type $\Gamma \vdash e : \tau$.
   (b) *If* $\Delta\ [\tau'/t]\Gamma \vdash [\tau'/t]r : [\tau'/t]\tau \Mapsto [\tau'/t]\tau''$ *and* $\Delta \vdash \tau'$ type *then* Δ, t type $\Gamma \vdash r : \tau \Mapsto \tau''$.
3. (a) *If* $\Delta\ \Gamma \vdash [e'/x]e : \tau$ *and* $\Delta\ \Gamma \vdash e' : \tau'$ *then* $\Delta\ \Gamma, x : \tau' \vdash e : \tau$.
   (b) *If* $\Delta\ \Gamma \vdash [e'/x]r : \tau \Mapsto \tau''$ *and* $\Delta\ \Gamma \vdash e' : \tau'$ *then* $\Delta\ \Gamma, x : \tau' \vdash r : \tau \Mapsto \tau''$.

Proof Sketch.

1. By rule induction over Rules (B.1) and case analysis over the definition of substitution. In all cases, the derivation of $\Delta \vdash [\tau'/t]\tau$ type does not depend on the form of τ'.
2. By mutual rule induction over Rules (B.2) and Rule (B.3) and case analysis over the definition of substitution. In all cases, the derivation of $\Delta\ [\tau'/t]\Gamma \vdash [\tau'/t]e : [\tau'/t]\tau$ or $\Delta\ [\tau'/t]\Gamma \vdash [\tau'/t]r : [\tau'/t]\tau \Rrightarrow [\tau'/t]\tau''$ does not depend on the form of τ'.
3. By mutual rule induction over Rules (B.2) and Rule (B.3) and case analysis over the definition of substitution. In all cases, the derivation of $\Delta\ \Gamma \vdash [e'/x]e : \tau$ or $\Delta\ \Gamma \vdash [e'/x]r : \tau \Rrightarrow \tau''$ does not depend on the form of e'.

□

Lemma B.5 (Pattern Regularity). *If* $\Delta \vdash p : \tau \dashv\!\mid \Gamma$ *and* $\Delta \vdash \tau$ type *then* $\Delta \vdash \Gamma$ ctx *and* $\mathsf{patvars}(p) = dom(\Gamma)$.

Proof. By rule induction over Rules (B.4).

Case (B.4a).

(1)	$p = x$	by assumption
(2)	$\Gamma = x : \tau$	by assumption
(3)	$\Delta \vdash \tau$ type	by assumption
(4)	$\Delta \vdash x : \tau$ ctx	by Definition B.1 on (3)
(5)	$\mathsf{fv}(p) = \mathrm{dom}(\Gamma) = \{x\}$	by definition

Case (B.4b).

(1)	$p = \mathtt{wildp}$	by assumption
(2)	$\Gamma = \emptyset$	by assumption
(3)	$\Delta \vdash \emptyset$ ctx	by Definition B.1
(4)	$\mathsf{patvars}(p) = \mathrm{dom}(\Gamma) = \emptyset$	by definition

Case (B.4d).

(1)	$p = \mathtt{tplp}[L](\{i \hookrightarrow p_i\}_{i\in L})$	by assumption
(2)	$\tau = \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})$	by assumption
(3)	$\Gamma = \cup_{i\in L}\Gamma_i$	by assumption
(4)	$\{\Delta \vdash p_i : \tau_i \dashv\!\mid \Gamma_i\}_{i\in L}$	by assumption
(5)	$\Delta \vdash \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})$ type	by assumption
(6)	$\{\Delta \vdash \tau_i \text{ type}\}_{i\in L}$	by Inversion of Rule (B.1e) on (5)
(7)	$\{\Delta \vdash \Gamma_i \text{ ctx}\}_{i\in L}$	by IH over (4) and (6)
(8)	$\{\mathsf{patvars}(p_i) = \mathrm{dom}(\Gamma_i)\}_{i\in L}$	by IH over (4) and (6)
(9)	$\Delta \vdash \cup_{i\in L}\Gamma_i$ ctx	by Definition B.1 over (7), then Definition B.1 iteratively
(10)	$\mathsf{patvars}(p) = \mathrm{dom}(\Gamma) = \emptyset$	by definition and (8)

Case (B.4e).

(1)	$p = \mathtt{injp}[\ell](p')$	by assumption
(2)	$\tau = \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau')$	by assumption
(3)	$\Delta \vdash \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau')$ type	by assumption
(4)	$\Delta \vdash p' : \tau' \dashv\!\!\mid \Gamma$	by assumption
(5)	$\Delta \vdash \tau'$ type	by Inversion of Rule (B.1f) on (3)
(6)	$\Delta \vdash \Gamma$ ctx	by IH on (4) and (5)
(7)	$\mathsf{patvars}(p') = \mathrm{dom}(\Gamma)$	by IH on (4) and (5)
(8)	$\mathsf{patvars}(p) = \mathrm{dom}(\Gamma)$	by definition and (7)

□

B.1.3 Structural Dynamics

The *structural dynamics* is specified as a transition system, and is organized around judgements of the following form:

Judgement Form	**Description**
$e \mapsto e'$	e transitions to e'
e val	e is a value
e matchfail	e raises match failure

We also define auxiliary judgements for *iterated transition*, $e \mapsto^* e'$, and *evaluation*, $e \Downarrow e'$.

Definition B.6 (Iterated Transition). *Iterated transition, $e \mapsto^* e'$, is the reflexive, transitive closure of the transition judgement, $e \mapsto e'$.*

Definition B.7 (Evaluation). *$e \Downarrow e'$ iff $e \mapsto^* e'$ and e'* val.

Our subsequent developments do not make mention of particular rules in the dynamics, nor do they make mention of other judgements, not listed above, that are used only for defining the dynamics of the match operator, so we do not produce these details here. Instead, it suffices to state the following conditions.

Condition B.8 (Canonical Forms). *If $\vdash e : \tau$ and e* val *then:*

1. *If $\tau = \mathtt{parr}(\tau_1;\tau_2)$ then $e = \mathtt{lam}\{\tau_1\}(x.e')$ and $x : \tau_1 \vdash e' : \tau_2$.*
2. *If $\tau = \mathtt{all}(t.\tau')$ then $e = \mathtt{tlam}(t.e')$ and t type $\vdash e' : \tau'$.*
3. *If $\tau = \mathtt{rec}(t.\tau')$ then $e = \mathtt{fold}(e')$ and $\vdash e' : [\mathtt{rec}(t.\tau')/t]\tau'$ and e'* val.
4. *If $\tau = \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})$ then $e = \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L})$ and $\vdash e_i : \tau_i$ and e_i* val *for each $i \in L$.*
5. *If $\tau = \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L})$ then for some label set L' and label ℓ and type τ', we have that $L = L', \ell$ and $\tau = \mathtt{sum}[L', \ell](\{i \hookrightarrow \tau_i\}_{i \in L'}; \ell \hookrightarrow \tau')$ and $e = \mathtt{inj}[\ell](e')$ and $\vdash e' : \tau'$ and e'* val.

Condition B.9 (Preservation). *If $\vdash e : \tau$ and $e \mapsto e'$ then $\vdash e' : \tau$.*

Condition B.10 (Progress). *If $\vdash e : \tau$ then either e* val *or e* matchfail *or there exists an e' such that $e \mapsto e'$.*

B.2 Unexpanded Language (UL)

B.2.1 Syntax

Stylized Syntax

Sort			Stylized Form	Description
UTyp	$\hat{\tau}$	$::=$	$\hat{t}$	identifier
			$\hat{\tau} \rightharpoonup \hat{\tau}$	partial function
			$\forall \hat{t}.\hat{\tau}$	polymorphic
			$\mu \hat{t}.\hat{\tau}$	recursive
			$\langle \{i \hookrightarrow \hat{\tau}_i\}_{i \in L} \rangle$	labeled product
			$[\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}]$	labeled sum
UExp	$\hat{e}$	$::=$	$\hat{x}$	identifier
			$\hat{e} : \hat{\tau}$	ascription
			$\texttt{let val}\ \hat{x} = \hat{e}\ \texttt{in}\ \hat{e}$	value binding
			$\lambda \hat{x}{:}\hat{\tau}.\hat{e}$	abstraction
			$\hat{e}(\hat{e})$	application
			$\Lambda \hat{t}.\hat{e}$	type abstraction
			$\hat{e}[\hat{\tau}]$	type application
			$\texttt{fold}(\hat{e})$	fold
			$\texttt{unfold}(\hat{e})$	unfold
			$\langle \{i \hookrightarrow \hat{e}_i\}_{i \in L} \rangle$	labeled tuple
			$\hat{e} \cdot \ell$	projection
			$\texttt{inj}[\ell](\hat{e})$	injection
			$\texttt{case}\ \hat{e}\ \{i \hookrightarrow \hat{x}_i.\hat{e}_i\}_{i \in L}$	case analysis
			$\texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{by static}\ e\ \texttt{in}\ \hat{e}$	seTSM definition
			$\hat{a}$ ‘b‘	seTSM application
			$\texttt{match}\ \hat{e}\ \{\hat{r}_i\}_{1 \leq i \leq n}$	match
			$\texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for patterns by static}\ e\ \texttt{in}\ \hat{e}$	spTSM definition
URule	$\hat{r}$	$::=$	$\hat{p} \Rightarrow \hat{e}$	match rule
UPat	$\hat{p}$	$::=$	$\hat{x}$	identifier pattern
			_	wildcard pattern
			$\texttt{fold}(\hat{p})$	fold pattern
			$\langle \{i \hookrightarrow \hat{p}_i\}_{i \in L} \rangle$	labeled tuple pattern
			$\texttt{inj}[\ell](\hat{p})$	injection pattern
			$\hat{a}$ ‘b‘	spTSM application

Body Lengths We write $\|b\|$ for the length of b. The metafunction $\|\hat{e}\|$ computes the sum of the lengths of expression literal bodies in $\hat{e}$:

$$
\begin{array}{ll}
\|\hat{x}\| & = 0 \\
\|\hat{e} : \hat{\tau}\| & = \|\hat{e}\| \\
\|\mathtt{let}\,\mathtt{val}\ \hat{x} = \hat{e}_1\ \mathtt{in}\ \hat{e}_2\| & = \|\hat{e}_1\| + \|\hat{e}_2\| \\
\|\lambda\hat{x}{:}\hat{\tau}.\hat{e}\| & = \|\hat{e}\| \\
\|\hat{e}_1(\hat{e}_2)\| & = \|\hat{e}_1\| + \|\hat{e}_2\| \\
\|\Lambda\hat{t}.\hat{e}\| & = \|\hat{e}\| \\
\|\hat{e}[\hat{\tau}]\| & = \|\hat{e}\| \\
\|\mathtt{fold}(\hat{e})\| & = \|\hat{e}\| \\
\|\mathtt{unfold}(\hat{e})\| & = \|\hat{e}\| \\
\|\langle\{i \hookrightarrow \hat{e}_i\}_{i\in L}\rangle\| & = \sum_{i\in L}\|\hat{e}_i\| \\
\|\ell \cdot \hat{e}\| & = \|\hat{e}\| \\
\|\mathtt{inj}[\ell](\hat{e})\| & = \|\hat{e}\| \\
\|\mathtt{case}\ \hat{e}\ \{i \hookrightarrow \hat{x}_i.\hat{e}_i\}_{i\in L}\| & = \|\hat{e}\| + \sum_{i\in L}\|\hat{e}_i\| \\
\|\mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{by}\ \mathtt{static}\ e_{\text{parse}}\ \mathtt{in}\ \hat{e}\| & = \|\hat{e}\| \\
\|\hat{a}\ `b`\| & = \|b\| \\
\|\mathtt{match}\ \hat{e}\ \{\hat{r}_i\}_{1\le i\le n}\| & = \|\hat{e}\| + \sum_{1\le i\le n}\|r_i\| \\
\|\mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{for}\ \mathtt{patterns}\ \mathtt{by}\ \mathtt{static}\ e_{\text{parse}}\ \mathtt{in}\ \hat{e}\| & = \|\hat{e}\|
\end{array}
$$

and $\|\hat{r}\|$ computes the sum of the lengths of expression literal bodies in $\hat{r}$:

$$\|\hat{p} \Rightarrow \hat{e}\| = \|\hat{e}\|$$

Similarly, the metafunction $\|\hat{p}\|$ computes the sum of the lengths of the pattern literal bodies in $\hat{p}$:

$$
\begin{aligned}
\|\hat{x}\| &= 0 \\
\|\mathtt{fold}(\hat{p})\| &= \|\hat{p}\| \\
\|\langle\{i \hookrightarrow \hat{p}_i\}_{i\in L}\rangle\| &= \sum_{i\in L}\|\hat{p}_i\| \\
\|\mathtt{inj}[\ell](\hat{p})\| &= \|\hat{p}\| \\
\|\hat{a}\ `b`\| &= \|b\|
\end{aligned}
$$

Common Unexpanded Forms Each expanded form maps onto an unexpanded form. We refer to these as the *common forms*. In particular:

- Each type variable, t, maps onto a unique type identifier, written $\widehat{t}$.
- Each type, τ, maps onto an unexpanded type, $\mathcal{U}(\tau)$, as follows:

$$
\begin{aligned}
\mathcal{U}(t) &= \widehat{t} \\
\mathcal{U}(\mathtt{parr}(\tau_1;\tau_2)) &= \mathcal{U}(\tau_1) \rightharpoonup \mathcal{U}(\tau_2) \\
\mathcal{U}(\mathtt{all}(t.\tau)) &= \forall\widehat{t}.\mathcal{U}(\tau)
\end{aligned}
$$

$$\mathcal{U}(\texttt{rec}(t.\tau)) = \mu\widehat{t}.\mathcal{U}(\tau)$$
$$\mathcal{U}(\texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})) = \langle\{i \hookrightarrow \mathcal{U}(\tau_i)\}_{i\in L}\rangle$$
$$\mathcal{U}(\texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i\in L})) = [\{i \hookrightarrow \mathcal{U}(\tau_i)\}_{i\in L}]$$

- Each expression variable, x, maps onto a unique expression identifier, written $\widehat{x}$.
- Each expanded expression, e, maps onto an unexpanded expression, $\mathcal{U}(e)$, as follows:

$$\begin{aligned}
\mathcal{U}(x) &= \widehat{x}\\
\mathcal{U}(\texttt{lam}\{\tau\}(x.e)) &= \lambda\widehat{x}{:}\mathcal{U}(\tau).\mathcal{U}(e)\\
\mathcal{U}(\texttt{ap}(e_1;e_2)) &= \mathcal{U}(e_1)(\mathcal{U}(e_2))\\
\mathcal{U}(\texttt{tlam}(t.e)) &= \Lambda\widehat{t}.\mathcal{U}(e)\\
\mathcal{U}(\texttt{tap}\{\tau\}(e)) &= \mathcal{U}(e)[\mathcal{U}(\tau)]\\
\mathcal{U}(\texttt{fold}(e)) &= \texttt{fold}(\mathcal{U}(e))\\
\mathcal{U}(\texttt{unfold}(e)) &= \texttt{unfold}(\mathcal{U}(e))\\
\mathcal{U}(\texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L})) &= \langle\{i \hookrightarrow \mathcal{U}(e_i)\}_{i\in L}\rangle\\
\mathcal{U}(\texttt{prj}[\ell](e)) &= \mathcal{U}(e)\cdot\ell\\
\mathcal{U}(\texttt{inj}[\ell](e)) &= \texttt{inj}[\ell](\mathcal{U}(e))\\
\mathcal{U}(\texttt{match}[n](e;\{r_i\}_{1\le i\le n})) &= \texttt{match}\,\mathcal{U}(e)\ \{\mathcal{U}(r_i)\}_{1\le i\le n}
\end{aligned}$$

- Each expanded rule, r, maps onto an unexpanded rule, $\mathcal{U}(r)$, as follows:

$$\mathcal{U}(\texttt{rule}(p.e)) = \texttt{urule}(\mathcal{U}(p).\mathcal{U}(e))$$

- Each expanded pattern, p, maps onto the unexpanded pattern, $\mathcal{U}(p)$, as follows:

$$\begin{aligned}
\mathcal{U}(x) &= \widehat{x}\\
\mathcal{U}(\texttt{wildp}) &= \texttt{uwildp}\\
\mathcal{U}(\texttt{foldp}(p)) &= \texttt{ufoldp}(\mathcal{U}(p))\\
\mathcal{U}(\texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i\in L})) &= \texttt{utplp}[L](\{i \hookrightarrow \mathcal{U}(p_i)\}_{i\in L})\\
\mathcal{U}(\texttt{injp}[\ell](p)) &= \texttt{uinjp}[\ell](\mathcal{U}(p))
\end{aligned}$$

Textual Syntax

In addition to the stylized syntax, there is also a context-free textual syntax for the UL. For our purposes, we need only posit the existence of partial metafunctions $\mathsf{parseUTyp}(b)$ and $\mathsf{parseUExp}(b)$ and $\mathsf{parseUPat}(b)$.

Condition B.11 (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 $\mathsf{parseUPat}(b) = \hat{p}$.*

We also impose the following technical conditions.

Condition B.12 (Expression Parsing Monotonicity)**.** *If $\mathsf{parseUExp}(b) = \hat{e}$ then $\|\hat{e}\| < \|b\|$.*

Condition B.13 (Pattern Parsing Monotonicity)**.** *If* $\mathsf{parseUPat}(b) = \hat{p}$ *then* $\|\hat{p}\| < \|b\|$.

B.2.2 Type Expansion

Unexpanded type formation contexts, $\hat{\Delta}$, are of the form $\langle \mathcal{D}; \Delta \rangle$, i.e. they consist of a *type identifier expansion context*, $\mathcal{D}$, paired with a type formation context, Δ.

A *type identifier expansion context*, $\mathcal{D}$, is a finite function that maps each type identifier $\hat{t} \in \text{dom}(\mathcal{D})$ to the hypothesis $\hat{t} \rightsquigarrow t$, for some type variable t. We write $\mathcal{D} \uplus \hat{t} \rightsquigarrow t$ for the type identifier expansion context that maps $\hat{t}$ to $\hat{t} \rightsquigarrow t$ and defers to $\mathcal{D}$ for all other type identifiers (i.e. the previous mapping is *updated*.)

We define $\hat{\Delta}, \hat{t} \rightsquigarrow t \;\mathsf{type}$ when $\hat{\Delta} = \langle \mathcal{D}; \Delta \rangle$ as an abbreviation of

$$\langle \mathcal{D} \uplus \hat{t} \rightsquigarrow t; \Delta, t \;\mathsf{type} \rangle$$

Definition B.14 (Unexpanded Type Formation Context Formation)**.** $\vdash \langle \mathcal{D}; \Delta \rangle \;\mathsf{utctx}$ *iff for each* $\hat{t} \rightsquigarrow t \;\mathsf{type} \in \mathcal{D}$ *we have* $t \;\mathsf{type} \in \Delta$.

$\boxed{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \;\mathsf{type}}$ $\hat{\tau}$ has well-formed expansion τ

$$\frac{}{\hat{\Delta}, \hat{t} \rightsquigarrow t \;\mathsf{type} \vdash \hat{t} \rightsquigarrow t \;\mathsf{type}} \tag{B.5a}$$

$$\frac{\hat{\Delta} \vdash \hat{\tau}_1 \rightsquigarrow \tau_1 \;\mathsf{type} \qquad \hat{\Delta} \vdash \hat{\tau}_2 \rightsquigarrow \tau_2 \;\mathsf{type}}{\hat{\Delta} \vdash \mathtt{uparr}(\hat{\tau}_1; \hat{\tau}_2) \rightsquigarrow \mathtt{parr}(\tau_1; \tau_2) \;\mathsf{type}} \tag{B.5b}$$

$$\frac{\hat{\Delta}, \hat{t} \rightsquigarrow t \;\mathsf{type} \vdash \hat{\tau} \rightsquigarrow \tau \;\mathsf{type}}{\hat{\Delta} \vdash \mathtt{uall}(\hat{t}.\hat{\tau}) \rightsquigarrow \mathtt{all}(t.\tau) \;\mathsf{type}} \tag{B.5c}$$

$$\frac{\hat{\Delta}, \hat{t} \rightsquigarrow t \;\mathsf{type} \vdash \hat{\tau} \rightsquigarrow \tau \;\mathsf{type}}{\hat{\Delta} \vdash \mathtt{urec}(\hat{t}.\hat{\tau}) \rightsquigarrow \mathtt{rec}(t.\tau) \;\mathsf{type}} \tag{B.5d}$$

$$\frac{\{\hat{\Delta} \vdash \hat{\tau}_i \rightsquigarrow \tau_i \;\mathsf{type}\}_{i \in L}}{\hat{\Delta} \vdash \mathtt{uprod}[L](\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}) \rightsquigarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \;\mathsf{type}} \tag{B.5e}$$

$$\frac{\{\hat{\Delta} \vdash \hat{\tau}_i \rightsquigarrow \tau_i \;\mathsf{type}\}_{i \in L}}{\hat{\Delta} \vdash \mathtt{usum}[L](\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}) \rightsquigarrow \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \;\mathsf{type}} \tag{B.5f}$$

B.2.3 Typed Expression Expansion

Contexts

Unexpanded typing contexts, $\hat{\Gamma}$, are, similarly, of the form $\langle \mathcal{G}; \Gamma \rangle$, where $\mathcal{G}$ is an *expression identifier expansion context*, and Γ is a typing context. An expression identifier expansion context, $\mathcal{G}$, is a finite function that maps each expression identifier $\hat{x} \in \text{dom}(\mathcal{G})$ to the hypothesis $\hat{x} \rightsquigarrow x$, for some expression variable, x. 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.)

We define $\hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau$ when $\hat{\Gamma} = \langle \mathcal{G}; \Gamma \rangle$ as an abbreviation of

$$\langle \mathcal{G} \uplus \hat{x} \rightsquigarrow x; \Gamma, x : \tau \rangle$$

Definition B.15 (Unexpanded Typing Context Formation). *$\Delta \vdash \langle \mathcal{G}; \Gamma \rangle$ uctx iff $\Delta \vdash \Gamma$ ctx and for each $\hat{x} \rightsquigarrow x \in \mathcal{G}$, we have $x \in dom(\Gamma)$.*

Body Encoding and Decoding

An assumed 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* $b \downarrow_{\mathsf{Body}} e_{\text{body}}$. An inverse mapping is defined by the *body decoding judgement* $e_{\text{body}} \uparrow_{\mathsf{Body}} b$.

Judgement Form	Description
$b \downarrow_{\mathsf{Body}} e$	b has encoding e
$e \uparrow_{\mathsf{Body}} b$	e has decoding b

The following condition establishes an isomorphism between literal bodies and values of type `Body` mediated by the judgements above.

Condition B.16 (Body Isomorphism).

1. *For every literal body b, we have that $b \downarrow_{\mathsf{Body}} e_{body}$ for some e_{body} such that $\vdash e_{body} : \texttt{Body}$ and e_{body} val.*
2. *If $\vdash e_{body} : \texttt{Body}$ and e_{body} val then $e_{body} \uparrow_{\mathsf{Body}} b$ for some b.*
3. *If $b \downarrow_{\mathsf{Body}} e_{body}$ then $e_{body} \uparrow_{\mathsf{Body}} b$.*
4. *If $\vdash e_{body} : \texttt{Body}$ and e_{body} val and $e_{body} \uparrow_{\mathsf{Body}} b$ then $b \downarrow_{\mathsf{Body}} e_{body}$.*
5. *If $b \downarrow_{\mathsf{Body}} e_{body}$ and $b \downarrow_{\mathsf{Body}} e'_{body}$ then $e_{body} = e'_{body}$.*
6. *If $\vdash e_{body} : \texttt{Body}$ and e_{body} val and $e_{body} \uparrow_{\mathsf{Body}} b$ and $e_{body} \uparrow_{\mathsf{Body}} b'$ then $b = b'$.*

We also assume a partial metafunction, $\mathsf{subseq}(b; m; n)$, which extracts a subsequence of b starting at position m and ending at position n, inclusive, where m and n are natural numbers. The following condition is technically necessary.

Condition B.17 (Body Subsequencing). *If $\mathsf{subseq}(b; m; n) = b'$ then $\|b'\| \leq \|b\|$.*

Parse Results

The type abbreviated `ParseResultSE`, and an auxiliary abbreviation used below, is defined as follows:

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

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

The type abbreviated ParseResultSP, and an auxiliary abbreviation used below, is defined as follows:

$$L_{\texttt{SP}} \stackrel{\text{def}}{=} \texttt{ParseError}, \texttt{SuccessP}$$

$$\texttt{ParseResultSE} \stackrel{\text{def}}{=} \texttt{sum}[L_{\texttt{SP}}](\texttt{ParseError} \hookrightarrow \langle\rangle, \texttt{SuccessP} \hookrightarrow \texttt{PrPat})$$

seTSM Contexts

seTSM contexts, $\hat{\Psi}$, are of the form $\langle \mathcal{A}; \Psi \rangle$, where $\mathcal{A}$ is a *TSM identifier expansion context* and Ψ is a *seTSM definition context*.

A *TSM identifier expansion context*, $\mathcal{A}$, is a finite function mapping each TSM identifier $\hat{a} \in \text{dom}(\mathcal{A})$ to the *TSM identifier expansion*, $\hat{a} \rightsquigarrow a$, for some *TSM name*, a. We write $\mathcal{A} \uplus \hat{a} \rightsquigarrow a$ for the TSM identifier expansion context that maps $\hat{a}$ to $\hat{a} \rightsquigarrow a$, and defers to $\mathcal{A}$ for all other TSM identifiers (i.e. the previous mapping is *updated*.)

An *seTSM definition context*, Ψ, is a finite function mapping each TSM name $a \in \text{dom}(\Psi)$ to an *expanded seTSM definition*, $a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}})$, where τ is the seTSM's type annotation, and e_{parse} is its parse function. We write $\Psi, a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}})$ when $a \notin \text{dom}(\Psi)$ for the extension of Ψ that maps a to $a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}})$. We write $\Delta \vdash \Psi \ \mathsf{seTSMs}$ when all the type annotations in Ψ are well-formed assuming Δ, and the parse functions in Ψ are closed and of the appropriate type.

Definition B.18 (seTSM Definition Context Formation)**.** $\Delta \vdash \Psi \ \mathsf{seTSMs}$ *iff for each* $a \hookrightarrow \texttt{setsm}(\tau; e_{parse}) \in \Psi$*, we have* $\Delta \vdash \tau \ \mathsf{type}$ *and* $\emptyset\, \emptyset \vdash e_{parse} : \texttt{parr}(\texttt{Body}; \texttt{ParseResultSE})$*.*

Definition B.19 (seTSM Context Formation)**.** $\Delta \vdash \langle \mathcal{A}; \Psi \rangle \ \mathsf{seTSMctx}$ *iff* $\Delta \vdash \Psi \ \mathsf{seTSMs}$ *and for each* $\hat{a} \rightsquigarrow a \in \mathcal{A}$ *we have* $a \in \text{dom}(\Psi)$*.*

We define $\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}})$, when $\hat{\Psi} = \langle \mathcal{A}; \Phi \rangle$, as an abbreviation of

$$\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}}) \rangle$$

spTSM Contexts

spTSM contexts, $\hat{\Phi}$, are of the form $\langle \mathcal{A}; \Phi \rangle$, where $\mathcal{A}$ is a TSM identifier expansion context, defined above, and Φ is a *spTSM definition context*.

An *spTSM definition context*, Φ, is a finite function mapping each TSM name $a \in \text{dom}(\Phi)$ to an *expanded seTSM definition*, $a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}})$, where τ is the spTSM's type annotation, and e_{parse} is its parse function. We write $\Phi, a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}})$ when $a \notin \text{dom}(\Phi)$ for the extension of Φ that maps a to $a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}})$. We write $\Delta \vdash \Phi \ \mathsf{spTSMs}$ when all the type annotations in Φ are well-formed assuming Δ, and the parse functions in Φ are closed and of the appropriate type.

Definition B.20 (spTSM Definition Context Formation)**.** $\Delta \vdash \Phi \ \mathsf{spTSMs}$ *iff for each* $a \hookrightarrow \texttt{sptsm}(\tau; e_{parse}) \in \Phi$*, we have* $\Delta \vdash \tau \ \mathsf{type}$ *and* $\emptyset\, \emptyset \vdash e_{parse} : \texttt{parr}(\texttt{Body}; \texttt{ParseResultSP})$*.*

Definition B.21 (spTSM Context Formation)**.** $\Delta \vdash \langle \mathcal{A}; \Phi \rangle \ \mathsf{spTSMctx}$ *iff* $\Delta \vdash \Phi \ \mathsf{spTSMs}$ *and for each* $\hat{a} \rightsquigarrow a \in \mathcal{A}$ *we have* $a \in \text{dom}(\Phi)$*.*

We define $\hat{\Phi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}})$, when $\hat{\Phi} = \langle \mathcal{A}; \Phi \rangle$, as an abbreviation of

$$\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}}) \rangle$$

Typed Expression Expansion

$\boxed{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e : \tau}$ $\hat{e}$ has expansion e of type τ

$$\frac{}{\hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{x} \rightsquigarrow x : \tau} \tag{B.6a}$$

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \text{ type} \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e : \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} : \hat{\tau} \rightsquigarrow e : \tau} \tag{B.6b}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e}_1 \rightsquigarrow e_1 : \tau_1 \qquad \hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau_1 \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e}_2 \rightsquigarrow e_2 : \tau_2}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{let}\,\texttt{val}\ \hat{x} = \hat{e}_1\ \texttt{in}\ \hat{e}_2 \rightsquigarrow \texttt{ap}(\texttt{lam}\{\tau_1\}(x.e_2); e_1) : \tau_2} \tag{B.6c}$$

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \text{ type} \qquad \hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e : \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \lambda \hat{x}{:}\hat{\tau}.\hat{e} \rightsquigarrow \texttt{lam}\{\tau\}(x.e) : \texttt{parr}(\tau; \tau')} \tag{B.6d}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e}_1 \rightsquigarrow e_1 : \texttt{parr}(\tau; \tau') \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e}_2 \rightsquigarrow e_2 : \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e}_1(\hat{e}_2) \rightsquigarrow \texttt{ap}(e_1; e_2) : \tau'} \tag{B.6e}$$

$$\frac{\hat{\Delta}, \hat{t} \rightsquigarrow t \text{ type}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e : \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \Lambda \hat{t}.\hat{e} \rightsquigarrow \texttt{tlam}(t.e) : \texttt{all}(t.\tau)} \tag{B.6f}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e : \texttt{all}(t.\tau) \qquad \hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau' \text{ type}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e}[\hat{\tau}'] \rightsquigarrow \texttt{tap}\{\tau'\}(e) : [\tau'/t]\tau} \tag{B.6g}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e : [\texttt{rec}(t.\tau)/t]\tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{fold}(\hat{e}) \rightsquigarrow \texttt{fold}(e) : \texttt{rec}(t.\tau)} \tag{B.6h}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e : \texttt{rec}(t.\tau)}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{unfold}(\hat{e}) \rightsquigarrow \texttt{unfold}(e) : [\texttt{rec}(t.\tau)/t]\tau} \tag{B.6i}$$

$$\frac{\{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e}_i \rightsquigarrow e_i : \tau_i\}_{i \in L}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \langle \{i \hookrightarrow \hat{e}_i\}_{i \in L} \rangle \rightsquigarrow \texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{B.6j}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e : \texttt{prod}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \cdot \ell \rightsquigarrow \texttt{prj}[\ell](e) : \tau} \tag{B.6k}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{inj}[\ell](\hat{e}) \rightsquigarrow \texttt{inj}[\ell](e) : \texttt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i\in L}; \ell \hookrightarrow \tau')} \tag{B.6l}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i\in L}) \qquad \{\hat{\Delta}\ \hat{\Gamma}, \hat{x}_i \rightsquigarrow x_i : \tau_i \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_i \rightsquigarrow e_i : \tau\}_{i\in L}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{case}\ \hat{e}\ \{i \hookrightarrow \hat{x}_i.\hat{e}_i\}_{i\in L} \rightsquigarrow \texttt{case}[L](e; \{i \hookrightarrow x_i.e_i\}_{i\in L}) : \tau} \tag{B.6m}$$

$$\frac{\begin{array}{c}\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \emptyset\ \emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResultSE)} \\ e_{\text{parse}} \Downarrow e'_{\text{parse}} \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi},\hat{a}\rightsquigarrow a \hookrightarrow \texttt{setsm}(\tau;e'_{\text{parse}})} \hat{e} \rightsquigarrow e : \tau'\end{array}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{by static}\ e_{\text{parse}}\ \texttt{in}\ \hat{e} \rightsquigarrow e : \tau'} \tag{B.6n}$$

$$\frac{\begin{array}{c}\hat{\Psi} = \hat{\Psi}', \hat{a} \rightsquigarrow a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}}) \\ b \downarrow_{\mathsf{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj[SuccessE]}(e_{\text{proto}}) \qquad e_{\text{proto}} \uparrow_{\mathsf{PrExpr}} \grave{e} \\ \mathsf{seg}(\grave{e})\ \mathsf{segments}\ b \qquad \emptyset\ \emptyset \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e : \tau\end{array}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{a}\ `b` \rightsquigarrow e : \tau} \tag{B.6o}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau \qquad \{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \rightsquigarrow r_i : \tau \Mapsto \tau'\}_{1\le i\le n}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{match}\ \hat{e}\ \{\hat{r}_i\}_{1\le i\le n} \rightsquigarrow \texttt{match}[n](e; \{r_i\}_{1\le i\le n}) : \tau'} \tag{B.6p}$$

$$\frac{\begin{array}{c}\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \emptyset\ \emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResultSP)} \\ e_{\text{parse}} \Downarrow e'_{\text{parse}} \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi},\hat{a}\rightsquigarrow a \hookrightarrow \texttt{sptsm}(\tau;e'_{\text{parse}})} \hat{e} \rightsquigarrow e : \tau'\end{array}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for patterns by static}\ e_{\text{parse}}\ \texttt{in}\ \hat{e} \rightsquigarrow e : \tau'} \tag{B.6q}$$

$\boxed{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \rightsquigarrow r : \tau \Mapsto \tau'}$ $\hat{r}$ has expansion r taking values of type τ to values of type τ'

$$\frac{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\vert\ \langle \mathcal{G}'; \Gamma' \rangle \qquad \hat{\Delta}\ \langle \mathcal{G} \uplus \mathcal{G}'; \Gamma \cup \Gamma' \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau'}{\hat{\Delta}\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{urule}(\hat{p}.\hat{e}) \rightsquigarrow \texttt{rule}(p.e) : \tau \Mapsto \tau'} \tag{B.7}$$

Typed Pattern Expansion

$\boxed{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\vert\ \hat{\Gamma}}$ $\hat{p}$ has expansion p matching against τ generating hypotheses $\hat{\Gamma}$

$$\frac{}{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{x} \rightsquigarrow x : \tau \dashv\!\vert\ \langle \hat{x} \rightsquigarrow x; x : \tau \rangle} \tag{B.8a}$$

$$\frac{}{\hat{\Delta} \vdash_{\hat{\Phi}} \_ \rightsquigarrow \texttt{wildp} : \tau \dashv\!\vert\ \langle \emptyset; \emptyset \rangle} \tag{B.8b}$$

$$\frac{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : [\texttt{rec}(t.\tau)/t]\tau \dashv\!\mid \hat{\Gamma}}{\hat{\Delta} \vdash_{\hat{\Phi}} \texttt{fold}(\hat{p}) \rightsquigarrow \texttt{foldp}(p) : \texttt{rec}(t.\tau) \dashv\!\mid \hat{\Gamma}} \tag{B.8c}$$

$$\frac{\begin{array}{c}\tau = \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \\ \{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p}_i \rightsquigarrow p_i : \tau_i \dashv\!\mid \hat{\Gamma}_i\}_{i \in L}\end{array}}{\hat{\Delta} \vdash_{\hat{\Phi}} \langle\{i \hookrightarrow \hat{p}_i\}_{i \in L}\rangle \rightsquigarrow \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \tau \dashv\!\mid \uplus_{i \in L} \hat{\Gamma}_i} \tag{B.8d}$$

$$\frac{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \hat{\Gamma}}{\hat{\Delta} \vdash_{\hat{\Phi}} \texttt{inj}[\ell](\hat{p}) \rightsquigarrow \texttt{injp}[\ell](p) : \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\!\mid \hat{\Gamma}} \tag{B.8e}$$

$$\frac{\begin{array}{c}\hat{\Phi} = \hat{\Phi}', \hat{a} \rightsquigarrow a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}}) \\ b \downarrow_{\mathsf{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}[\mathsf{SuccessP}](e_{\text{proto}}) \qquad e_{\text{proto}} \uparrow_{\mathsf{PrPat}} \grave{p} \\ \mathsf{seg}(\grave{p})\ \mathsf{segments}\ b \qquad \grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\hat{\Delta}; \hat{\Phi}; b} \hat{\Gamma}\end{array}}{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{a}\ \text{`}b\text{`} \rightsquigarrow p : \tau \dashv\!\mid \hat{\Gamma}} \tag{B.8f}$$

In Rule (B.8d), $\hat{\Gamma}_i$ is shorthand for $\langle \mathcal{G}_i; \Gamma_i \rangle$ and $\uplus_{i \in L} \hat{\Gamma}_i$ is shorthand for

$$\langle \uplus_{i \in L} \mathcal{G}_i; \cup_{i \in L} \Gamma_i \rangle$$

B.3 Proto-Expansion Validation

B.3.1 Syntax of Proto-Expansions

Sort		Operational Form	Stylized Form	Description
PrTyp	$\grave{\tau}$ ::=	t	t	variable
		$\texttt{prparr}(\grave{\tau};\grave{\tau})$	$\grave{\tau} \rightharpoonup \grave{\tau}$	partial function
		$\texttt{prall}(t.\grave{\tau})$	$\forall t.\grave{\tau}$	polymorphic
		$\texttt{prrec}(t.\grave{\tau})$	$\mu t.\grave{\tau}$	recursive
		$\texttt{prprod}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i\in L})$	$\langle\{i \hookrightarrow \grave{\tau}_i\}_{i\in L}\rangle$	labeled product
		$\texttt{prsum}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i\in L})$	$[\{i \hookrightarrow \grave{\tau}_i\}_{i\in L}]$	labeled sum
		$\texttt{splicedt}[m;n]$	$\texttt{splicedt}[m;n]$	spliced type ref.
PrExp	$\grave{e}$ ::=	x	x	variable
		$\texttt{prasc}\{\grave{\tau}\}(\grave{e})$	$\grave{e} : \grave{\tau}$	ascription
		$\texttt{prletval}(\grave{e};x.\grave{e})$	$\texttt{let}\,\texttt{val}\ x = \grave{e}\ \texttt{in}\ \grave{e}$	value binding
		$\texttt{prlam}\{\grave{\tau}\}(x.\grave{e})$	$\lambda x{:}\grave{\tau}.\grave{e}$	abstraction
		$\texttt{prap}(\grave{e};\grave{e})$	$\grave{e}(\grave{e})$	application
		$\texttt{prtlam}(t.\grave{e})$	$\Lambda t.\grave{e}$	type abstraction
		$\texttt{prtap}\{\grave{\tau}\}(\grave{e})$	$\grave{e}[\grave{\tau}]$	type application
		$\texttt{prfold}(\grave{e})$	$\texttt{fold}(\grave{e})$	fold
		$\texttt{prunfold}(\grave{e})$	$\texttt{unfold}(\grave{e})$	unfold
		$\texttt{prtpl}\{L\}(\{i \hookrightarrow \grave{e}_i\}_{i\in L})$	$\langle\{i \hookrightarrow \grave{e}_i\}_{i\in L}\rangle$	labeled tuple
		$\texttt{prprj}[\ell](\grave{e})$	$\grave{e}\cdot\ell$	projection
		$\texttt{prinj}[\ell](\grave{e})$	$\texttt{inj}[\ell](\grave{e})$	injection
		$\texttt{prcase}[L](\grave{e};\{i \hookrightarrow x_i.\grave{e}_i\}_{i\in L})$	$\texttt{case}\ \grave{e}\ \{i \hookrightarrow x_i.\grave{e}_i\}_{i\in L}$	case analysis
		$\texttt{splicede}[m;n;\grave{\tau}]$	$\texttt{splicede}[m;n;\grave{\tau}]$	spliced expr. ref.
		$\texttt{prmatch}[n](\grave{e};\{\grave{r}_i\}_{1\leq i\leq n})$	$\texttt{match}\ \grave{e}\ \{\grave{r}_i\}_{1\leq i\leq n}$	match
PrRule	$\grave{r}$ ::=	$\texttt{prrule}(p.\grave{e})$	$p \Rightarrow \grave{e}$	rule
PrPat	$\grave{p}$ ::=	$\texttt{prwildp}$	_	wildcard pattern
		$\texttt{prfoldp}(p)$	$\texttt{fold}(p)$	fold pattern
		$\texttt{prtplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i\in L})$	$\langle\{i \hookrightarrow \grave{p}_i\}_{i\in L}\rangle$	labeled tuple pattern
		$\texttt{prinjp}[\ell](\grave{p})$	$\texttt{inj}[\ell](\grave{p})$	injection pattern
		$\texttt{splicedp}[m;n;\grave{\tau}]$	$\texttt{splicedp}[m;n;\grave{\tau}]$	spliced pattern ref.

Common Proto-Expansion Terms

Each expanded term, except variable patterns, maps onto a proto-expansion term. We refer to these as the *common proto-expansion terms*. In particular:

- Each type, τ, maps onto a proto-type, $\mathcal{P}(\tau)$, as follows:

$$\begin{aligned}
\mathcal{P}(t) &= t \\
\mathcal{P}(\texttt{parr}(\tau_1;\tau_2)) &= \texttt{prparr}(\mathcal{P}(\tau_1);\mathcal{P}(\tau_2)) \\
\mathcal{P}(\texttt{all}(t.\tau)) &= \texttt{prall}(t.\mathcal{P}(\tau)) \\
\mathcal{P}(\texttt{rec}(t.\tau)) &= \texttt{prrec}(t.\mathcal{P}(\tau)) \\
\mathcal{P}(\texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})) &= \texttt{prprod}[L](\{i \hookrightarrow \mathcal{P}(\tau_i)\}_{i\in L}) \\
\mathcal{P}(\texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i\in L})) &= \texttt{prsum}[L](\{i \hookrightarrow \mathcal{P}(\tau_i)\}_{i\in L})
\end{aligned}$$

- Each expanded expression, e, maps onto a proto-expression, $\mathcal{P}(e)$, as follows:

$$\begin{aligned}
\mathcal{P}(x) &= x \\
\mathcal{P}(\texttt{lam}\{\tau\}(x.e)) &= \texttt{prlam}\{\mathcal{P}(\tau)\}(x.\mathcal{P}(e)) \\
\mathcal{P}(\texttt{ap}(e_1;e_2)) &= \texttt{prap}(\mathcal{P}(e_1);\mathcal{P}(e_2)) \\
\mathcal{P}(\texttt{tlam}(t.e)) &= \texttt{prtlam}(t.\mathcal{P}(e)) \\
\mathcal{P}(\texttt{tap}\{\tau\}(e)) &= \texttt{prtap}\{\mathcal{P}(\tau)\}(\mathcal{P}(e)) \\
\mathcal{P}(\texttt{fold}(e)) &= \texttt{prfold}(\mathcal{P}(e)) \\
\mathcal{P}(\texttt{unfold}(e)) &= \texttt{prunfold}(\mathcal{P}(e)) \\
\mathcal{P}(\texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L})) &= \texttt{prtpl}\{L\}(\{i \hookrightarrow \mathcal{P}(e_i)\}_{i\in L}) \\
\mathcal{P}(\texttt{inj}[\ell](e)) &= \texttt{prinj}[\ell](\mathcal{P}(e)) \\
\mathcal{P}(\texttt{match}[n](e;\{r_i\}_{1\leq i\leq n})) &= \texttt{prmatch}[n](\mathcal{P}(e);\{\mathcal{P}(r_i)\}_{1\leq i\leq n})
\end{aligned}$$

- Each expanded rule, r, maps onto the proto-rule, $\mathcal{P}(r)$, as follows:

$$\mathcal{P}(\texttt{rule}(p.e)) = \texttt{prrule}(p.\mathcal{P}(e))$$

  Notice that proto-rules bind expanded patterns, not proto-patterns. This is because proto-rules appear in proto-expressions, which are generated by seTSMs. It would not be sensible for an seTSM to splice a pattern out of a literal body.

- Each expanded pattern, p, except for the variable patterns, maps onto a proto-pattern, $\mathcal{P}(p)$, as follows:

$$\begin{aligned}
\mathcal{P}(\texttt{wildp}) &= \texttt{prwildp} \\
\mathcal{P}(\texttt{foldp}(p)) &= \texttt{prfoldp}(\mathcal{P}(p)) \\
\mathcal{P}(\texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i\in L})) &= \texttt{prtplp}[L](\{i \hookrightarrow \mathcal{P}(p_i)\}_{i\in L}) \\
\mathcal{P}(\texttt{injp}[\ell](p)) &= \texttt{prinjp}[\ell](\mathcal{P}(p))
\end{aligned}$$

Proto-Expression Encoding and Decoding

The type abbreviated `PrExpr` classifies encodings of *proto-expressions*. The mapping from proto-expressions to values of type `PrExpr` is defined by the *proto-expression encoding judgement*, $\grave{e} \downarrow_{\mathsf{PrExpr}} e$. An inverse mapping is defined by the *proto-expression decoding judgement*, $e \uparrow_{\mathsf{PrExpr}} \grave{e}$.

Judgement Form	Description
$\grave{e} \downarrow_{\mathsf{PrExpr}} e$	$\grave{e}$ has encoding e
$e \uparrow_{\mathsf{PrExpr}} \grave{e}$	e has decoding $\grave{e}$

Rather than picking a particular definition of `PrExpr` and defining the judgements above inductively against it, we only state the following condition, which establishes an isomorphism between values of type `PrExpr` and proto-expressions.

Condition B.22 (Proto-Expression Isomorphism)**.**

1. *For every $\grave{e}$, we have $\grave{e} \downarrow_{\mathsf{PrExpr}} e_{proto}$ for some e_{proto} such that $\vdash e_{proto} : \mathtt{PrExpr}$ and e_{proto} val.*
2. *If $\vdash e_{proto} : \mathtt{PrExpr}$ and e_{proto} val then $e_{proto} \uparrow_{\mathsf{PrExpr}} \grave{e}$ for some $\grave{e}$.*
3. *If $\grave{e} \downarrow_{\mathsf{PrExpr}} e_{proto}$ then $e_{proto} \uparrow_{\mathsf{PrExpr}} \grave{e}$.*
4. *If $\vdash e_{proto} : \mathtt{PrExpr}$ and e_{proto} val and $e_{proto} \uparrow_{\mathsf{PrExpr}} \grave{e}$ then $\grave{e} \downarrow_{\mathsf{PrExpr}} e_{proto}$.*
5. *If $\grave{e} \downarrow_{\mathsf{PrExpr}} e_{proto}$ and $\grave{e} \downarrow_{\mathsf{PrExpr}} e'_{proto}$ then $e_{proto} = e'_{proto}$.*
6. *If $\vdash e_{proto} : \mathtt{PrExpr}$ and e_{proto} val and $e_{proto} \uparrow_{\mathsf{PrExpr}} \grave{e}$ and $e_{proto} \uparrow_{\mathsf{PrExpr}} \grave{e}'$ then $\grave{e} = \grave{e}'$.*

Proto-Pattern Encoding and Decoding

The type abbreviated `PrPat` classifies encodings of *proto-patterns*. The mapping from proto-patterns to values of type `PrPat` is defined by the *proto-pattern encoding judgement*, $\grave{p} \downarrow_{\mathsf{PrPat}} p$. An inverse mapping is defined by the *proto-expression decoding judgement*, $p \uparrow_{\mathsf{PrPat}} \grave{p}$.

Judgement Form	Description
$\grave{p} \downarrow_{\mathsf{PrPat}} p$	$\grave{p}$ has encoding p
$p \uparrow_{\mathsf{PrPat}} \grave{p}$	p has decoding $\grave{p}$

Again, rather than picking a particular definition of `PrPat` and defining the judgements above inductively against it, we only state the following condition, which establishes an isomorphism between values of type `PrPat` and proto-patterns.

Condition B.23 (Proto-Pattern Isomorphism)**.**

1. *For every $\grave{p}$, we have $\grave{p} \downarrow_{\mathsf{PrPat}} e_{proto}$ for some e_{proto} such that $\vdash e_{proto} : \mathtt{PrPat}$ and e_{proto} val.*
2. *If $\vdash e_{proto} : \mathtt{PrPat}$ and e_{proto} val then $e_{proto} \uparrow_{\mathsf{PrPat}} \grave{p}$ for some $\grave{p}$.*
3. *If $\grave{p} \downarrow_{\mathsf{PrPat}} e_{proto}$ then $e_{proto} \uparrow_{\mathsf{PrPat}} \grave{p}$.*
4. *If $\vdash e_{proto} : \mathtt{PrPat}$ and e_{proto} val and $e_{proto} \uparrow_{\mathsf{PrPat}} \grave{p}$ then $\grave{p} \downarrow_{\mathsf{PrPat}} e_{proto}$.*
5. *If $\grave{p} \downarrow_{\mathsf{PrPat}} e_{proto}$ and $\grave{p} \downarrow_{\mathsf{PrPat}} e'_{proto}$ then $e_{proto} = e'_{proto}$.*
6. *If $\vdash e_{proto} : \mathtt{PrPat}$ and e_{proto} val and $e_{proto} \uparrow_{\mathsf{PrPat}} \grave{p}$ and $e_{proto} \uparrow_{\mathsf{PrPat}} \grave{p}'$ then $\grave{p} = \grave{p}'$.*

Splice Summaries

The *splice summary* of a proto-expression, $\mathsf{summary}(\grave{e})$, or proto-pattern, $\mathsf{summary}(\grave{p})$, is the finite set of references to spliced types, expressions and patterns that it mentions.

Segmentations

A *segment set*, ψ, is a finite set of pairs of natural numbers indicating the locations of spliced terms. The *segmentation* of a proto-expression, $\mathsf{seg}(\grave{e})$, or proto-pattern, $\mathsf{seg}(\grave{p})$, is the segment set implied by its splice summary.

The predicate ψ $\mathsf{segments}$ b checks that each segment in ψ, has non-negative length and is within bounds of b, and that the segments in ψ do not overlap.

B.3.2 Proto-Type Validation

Type splicing scenes, $\mathbb{T}$, are of the form $\hat{\Delta}$; b.

$\boxed{\Delta \vdash^{\mathbb{T}} \grave{\tau} \leadsto \tau \;\mathsf{type}}$ $\grave{\tau}$ has well-formed expansion τ

$$\frac{}{\Delta, t\;\mathsf{type} \vdash^{\mathbb{T}} t \leadsto t\;\mathsf{type}} \tag{B.9a}$$

$$\frac{\Delta \vdash^{\mathbb{T}} \grave{\tau}_1 \leadsto \tau_1\;\mathsf{type} \qquad \Delta \vdash^{\mathbb{T}} \grave{\tau}_2 \leadsto \tau_2\;\mathsf{type}}{\Delta \vdash^{\mathbb{T}} \mathtt{prparr}(\grave{\tau}_1; \grave{\tau}_2) \leadsto \mathtt{parr}(\tau_1; \tau_2)\;\mathsf{type}} \tag{B.9b}$$

$$\frac{\Delta, t\;\mathsf{type} \vdash^{\mathbb{T}} \grave{\tau} \leadsto \tau\;\mathsf{type}}{\Delta \vdash^{\mathbb{T}} \mathtt{prall}(t.\grave{\tau}) \leadsto \mathtt{all}(t.\tau)\;\mathsf{type}} \tag{B.9c}$$

$$\frac{\Delta, t\;\mathsf{type} \vdash^{\mathbb{T}} \grave{\tau} \leadsto \tau\;\mathsf{type}}{\Delta \vdash^{\mathbb{T}} \mathtt{prrec}(t.\grave{\tau}) \leadsto \mathtt{rec}(t.\tau)\;\mathsf{type}} \tag{B.9d}$$

$$\frac{\{\Delta \vdash^{\mathbb{T}} \grave{\tau}_i \leadsto \tau_i\;\mathsf{type}\}_{i \in L}}{\Delta \vdash^{\mathbb{T}} \mathtt{prprod}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}) \leadsto \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})\;\mathsf{type}} \tag{B.9e}$$

$$\frac{\{\Delta \vdash^{\mathbb{T}} \grave{\tau}_i \leadsto \tau_i\;\mathsf{type}\}_{i \in L}}{\Delta \vdash^{\mathbb{T}} \mathtt{prsum}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}) \leadsto \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L})\;\mathsf{type}} \tag{B.9f}$$

$$\frac{\mathsf{parseUTyp}(\mathsf{subseq}(b; m; n)) = \hat{\tau} \qquad \langle \mathcal{D}; \Delta_{\mathrm{app}} \rangle \vdash \hat{\tau} \leadsto \tau\;\mathsf{type} \qquad \Delta \cap \Delta_{\mathrm{app}} = \emptyset}{\Delta \vdash^{\langle \mathcal{D}; \Delta_{\mathrm{app}} \rangle; b} \mathtt{splicedt}[m; n] \leadsto \tau\;\mathsf{type}} \tag{B.9g}$$

B.3.3 Proto-Expression Validation

Expression splicing scenes, $\mathbb{E}$, are of the form $\hat{\Delta}$; $\hat{\Gamma}$; $\hat{\Psi}$;$\hat{\Phi}$; b. We write $\mathsf{ts}(\mathbb{E})$ for the type splicing scene constructed by dropping unnecessary contexts from $\mathbb{E}$:

$$\mathsf{ts}(\hat{\Delta};\, \hat{\Gamma};\, \hat{\Psi}; \hat{\Phi};\, b) = \hat{\Delta};\, b$$

$\boxed{\Delta\;\Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau}$ $\grave{e}$ has expansion e of type τ

$$\frac{}{\Delta\;\Gamma, x : \tau \vdash^{\mathbb{E}} x \leadsto x : \tau} \tag{B.10a}$$

$$\frac{\Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau} \leadsto \tau~\mathsf{type} \qquad \Delta~\Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau}{\Delta~\Gamma \vdash^{\mathbb{E}} \mathtt{prasc}\{\grave{\tau}\}(\grave{e}) \leadsto e : \tau} \tag{B.10b}$$

$$\frac{\Delta~\Gamma \vdash^{\mathbb{E}} \grave{e}_1 \leadsto e_1 : \tau_1 \qquad \Delta~\Gamma, x : \tau_1 \vdash^{\grave{e}_2} e_2 \leadsto \tau_2 :}{\Delta~\Gamma \vdash^{\mathbb{E}} \mathtt{prletval}(\grave{e}_1; x.\grave{e}_2) \leadsto \mathtt{ap}(\mathtt{lam}\{\tau_1\}(x.e_2); e_1) : \tau_2} \tag{B.10c}$$

$$\frac{\Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau} \leadsto \tau~\mathsf{type} \qquad \Delta~\Gamma, x : \tau \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau'}{\Delta~\Gamma \vdash^{\mathbb{E}} \mathtt{prlam}\{\grave{\tau}\}(x.\grave{e}) \leadsto \mathtt{lam}\{\tau\}(x.e) : \mathtt{parr}(\tau; \tau')} \tag{B.10d}$$

$$\frac{\Delta~\Gamma \vdash^{\mathbb{E}} \grave{e}_1 \leadsto e_1 : \mathtt{parr}(\tau; \tau') \qquad \Delta~\Gamma \vdash^{\mathbb{E}} \grave{e}_2 \leadsto e_2 : \tau}{\Delta~\Gamma \vdash^{\mathbb{E}} \mathtt{prap}(\grave{e}_1; \grave{e}_2) \leadsto \mathtt{ap}(e_1; e_2) : \tau'} \tag{B.10e}$$

$$\frac{\Delta, t~\mathsf{type}~\Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau}{\Delta~\Gamma \vdash^{\mathbb{E}} \mathtt{prtlam}(t.\grave{e}) \leadsto \mathtt{tlam}(t.e) : \mathtt{all}(t.\tau)} \tag{B.10f}$$

$$\frac{\Delta~\Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e : \mathtt{all}(t.\tau) \qquad \Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau}' \leadsto \tau'~\mathsf{type}}{\Delta~\Gamma \vdash^{\mathbb{E}} \mathtt{prtap}\{\grave{\tau}'\}(\grave{e}) \leadsto \mathtt{tap}\{\tau'\}(e) : [\tau'/t]\tau} \tag{B.10g}$$

$$\frac{\Delta~\Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e : [\mathtt{rec}(t.\tau)/t]\tau}{\Delta~\Gamma \vdash^{\mathbb{E}} \mathtt{prfold}(\grave{e}) \leadsto \mathtt{fold}(e) : \mathtt{rec}(t.\tau)} \tag{B.10h}$$

$$\frac{\Delta~\Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e : \mathtt{rec}(t.\tau)}{\Delta~\Gamma \vdash^{\mathbb{E}} \mathtt{prunfold}(\grave{e}) \leadsto \mathtt{unfold}(e) : [\mathtt{rec}(t.\tau)/t]\tau} \tag{B.10i}$$

$$\frac{\begin{array}{c}\tau = \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \\ \{\Delta~\Gamma \vdash^{\mathbb{E}} \grave{e}_i \leadsto e_i : \tau_i\}_{i \in L}\end{array}}{\Delta~\Gamma \vdash^{\mathbb{E}} \mathtt{prtpl}\{L\}(\{i \hookrightarrow \grave{e}_i\}_{i \in L}) \leadsto \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) : \tau} \tag{B.10j}$$

$$\frac{\Delta~\Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e : \mathtt{prod}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)}{\Delta~\Gamma \vdash^{\mathbb{E}} \mathtt{prprj}[\ell](\grave{e}) \leadsto \mathtt{prj}[\ell](e) : \tau} \tag{B.10k}$$

$$\frac{\Delta~\Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau'}{\Delta~\Gamma \vdash^{\mathbb{E}} \mathtt{prinj}[\ell](\grave{e}) \leadsto \mathtt{inj}[\ell](e) : \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau')} \tag{B.10l}$$

$$\frac{\Delta~\Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e : \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \qquad \{\Delta~\Gamma, x_i : \tau_i \vdash^{\mathbb{E}} \grave{e}_i \leadsto e_i : \tau\}_{i \in L}}{\Delta~\Gamma \vdash^{\mathbb{E}} \mathtt{prcase}[L](\grave{e}; \{i \hookrightarrow x_i.\grave{e}_i\}_{i \in L}) \leadsto \mathtt{case}[L](e; \{i \hookrightarrow x_i.e_i\}_{i \in L}) : \tau} \tag{B.10m}$$

$$\frac{\begin{array}{c}\emptyset \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau} \leadsto \tau~\mathsf{type} \qquad \mathbb{E} = \langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle; \langle \mathcal{G}; \Gamma_{\mathsf{app}} \rangle; \hat{\Psi}; \hat{\Phi}; b \\ \mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e} \qquad \langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle~\langle \mathcal{G}; \Gamma_{\mathsf{app}} \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \leadsto e : \tau \\ \Delta \cap \Delta_{\mathsf{app}} = \emptyset \qquad \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma_{\mathsf{app}}) = \emptyset\end{array}}{\Delta~\Gamma \vdash^{\mathbb{E}} \mathtt{splicede}[m; n; \grave{\tau}] \leadsto e : \tau} \tag{B.10n}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau \qquad \{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r}_i \rightsquigarrow r_i : \tau \Mapsto \tau'\}_{1 \le i \le n}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prmatch}[n](\grave{e}; \{\grave{r}_i\}_{1 \le i \le n}) \rightsquigarrow \mathtt{match}[n](e; \{r_i\}_{1 \le i \le n}) : \tau'} \tag{B.10o}$$

$\boxed{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r} \rightsquigarrow r : \tau \Mapsto \tau'}$ $\grave{r}$ has expansion r taking values of type τ to values of type τ'

$$\frac{p : \tau \dashv\!\mid \Gamma' \qquad \Delta\ \Gamma \cup \Gamma' \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prrule}(p.\grave{e}) \rightsquigarrow \mathtt{rule}(p.e) : \tau \Mapsto \tau'} \tag{B.11}$$

B.3.4 Proto-Pattern Validation

Pattern splicing scenes, $\mathbb{P}$, are of the form $\hat{\Delta}; \hat{\Phi}; b$.

$\boxed{\grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\mathbb{P}} \hat{\Gamma}}$ $\grave{p}$ has expansion p matching against τ generating hypotheses $\hat{\Gamma}$

$$\frac{}{\mathtt{prwildp} \rightsquigarrow \mathtt{wildp} : \tau \dashv\!\mid^{\mathbb{P}} \langle\emptyset;\emptyset\rangle} \tag{B.12a}$$

$$\frac{\grave{p} \rightsquigarrow p : [\mathtt{rec}(t.\tau)/t]\tau \dashv\!\mid^{\mathbb{P}} \hat{\Gamma}}{\mathtt{prfoldp}(\grave{p}) \rightsquigarrow \mathtt{foldp}(p) : \mathtt{rec}(t.\tau) \dashv\!\mid^{\mathbb{P}} \hat{\Gamma}} \tag{B.12b}$$

$$\frac{\begin{array}{c}\tau = \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \\ \{\grave{p}_i \rightsquigarrow p_i : \tau_i \dashv\!\mid^{\mathbb{P}} \hat{\Gamma}_i\}_{i \in L}\end{array}}{\mathtt{prtplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i \in L}) \rightsquigarrow \mathtt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \tau \dashv\!\mid^{\mathbb{P}} \uplus_{i \in L} \hat{\Gamma}_i} \tag{B.12c}$$

$$\frac{\grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\mathbb{P}} \hat{\Gamma}}{\mathtt{prinjp}[\ell](\grave{p}) \rightsquigarrow \mathtt{injp}[\ell](p) : \mathtt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\!\mid^{\mathbb{P}} \hat{\Gamma}} \tag{B.12d}$$

$$\frac{\emptyset \vdash^{\hat{\Delta}; b} \grave{\tau} \rightsquigarrow \tau \ \mathsf{type} \qquad \mathsf{parseUPat}(\mathsf{subseq}(b; m; n)) = \hat{p} \qquad \hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \hat{\Gamma}}{\mathtt{splicedp}[m; n; \grave{\tau}] \rightsquigarrow p : \tau \dashv\!\mid^{\hat{\Delta}; \hat{\Phi}; b} \hat{\Gamma}} \tag{B.12e}$$

B.4 Metatheory

B.4.1 Type Expansion

Lemma B.24 (Type Expansion). *If* $\langle\mathcal{D}; \Delta\rangle \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type}$ *then* $\Delta \vdash \tau\ \mathsf{type}$.
Proof. By rule induction over Rules (B.5). In each case, we apply the IH to or over each premise, then apply the corresponding type formation rule in Rules (B.1). ☐

Lemma B.25 (Proto-Type Validation). *If* $\Delta \vdash^{\langle\mathcal{D}; \Delta_{app}\rangle; b} \grave{\tau} \rightsquigarrow \tau\ \mathsf{type}$ *and* $\Delta \cap \Delta_{app} = \emptyset$ *then* $\Delta \cup \Delta_{app} \vdash \tau\ \mathsf{type}$.
Proof. By rule induction over Rules (B.9).

Case (B.9a).

(1) $\Delta = \Delta', t\ \mathsf{type}$ — by assumption

(2) $\grave{\tau} = t$ — by assumption

(3) $\tau = t$ — by assumption

(4) $\Delta', t\ \mathsf{type} \vdash t\ \mathsf{type}$ — by Rule (B.1a)

(5) $\Delta', t\ \mathsf{type} \cup \Delta_{\mathsf{app}} \vdash t\ \mathsf{type}$ — by Lemma B.2 over Δ_{app} to (4)

Case (B.9b).

(1) $\grave{\tau} = \mathtt{prparr}(\grave{\tau}_1; \grave{\tau}_2)$ — by assumption

(2) $\tau = \mathtt{parr}(\tau_1; \tau_2)$ — by assumption

(3) $\Delta \vdash^{\langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle; b} \grave{\tau}_1 \rightsquigarrow \tau_1\ \mathsf{type}$ — by assumption

(4) $\Delta \vdash^{\langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle; b} \grave{\tau}_2 \rightsquigarrow \tau_2\ \mathsf{type}$ — by assumption

(5) $\Delta \cup \Delta_{\mathsf{app}} \vdash \tau_1\ \mathsf{type}$ — by IH on (3)

(6) $\Delta \cup \Delta_{\mathsf{app}} \vdash \tau_2\ \mathsf{type}$ — by IH on (4)

(7) $\Delta \cup \Delta_{\mathsf{app}} \vdash \mathtt{parr}(\tau_1; \tau_2)\ \mathsf{type}$ — by Rule (B.1b) on (5) and (6)

Case (B.9c).

(1) $\grave{\tau} = \mathtt{prall}(t.\grave{\tau}')$ — by assumption

(2) $\tau = \mathtt{all}(t.\tau')$ — by assumption

(3) $\Delta, t\ \mathsf{type} \vdash^{\langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle; b} \grave{\tau}' \rightsquigarrow \tau'\ \mathsf{type}$ — by assumption

(4) $\Delta, t\ \mathsf{type} \cup \Delta_{\mathsf{app}} \vdash \tau'\ \mathsf{type}$ — by IH on (3)

(5) $\Delta \cup \Delta_{\mathsf{app}}, t\ \mathsf{type} \vdash \tau'\ \mathsf{type}$ — by exchange over Δ_{app} on (4)

(6) $\Delta \cup \Delta_{\mathsf{app}} \vdash \mathtt{all}(t.\tau')\ \mathsf{type}$ — by Rule (B.1c) on (5)

Case (B.9d).

(1) $\grave{\tau} = \mathtt{prrec}(t.\grave{\tau}')$ — by assumption

(2) $\tau = \mathtt{rec}(t.\tau')$ — by assumption

(3) $\Delta, t\ \mathsf{type} \vdash^{\Delta_{\mathsf{app}}; b} \grave{\tau}' \rightsquigarrow \tau'\ \mathsf{type}$ — by assumption

(4) $\Delta, t\ \mathsf{type} \cup \Delta_{\mathsf{app}} \vdash \tau'\ \mathsf{type}$ — by IH on (3)

(5) $\Delta \cup \Delta_{\mathsf{app}}, t\ \mathsf{type} \vdash \tau'\ \mathsf{type}$ — by exchange over Δ_{app} on (4)

(6) $\Delta \cup \Delta_{\mathsf{app}} \vdash \mathtt{rec}(t.\tau')\ \mathsf{type}$ — by Rule (B.1d) on (5)

Case (B.9e).

(1) $\grave{\tau} = \mathtt{prprod}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L})$ — by assumption

(2) $\tau = \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})$ — by assumption

(3) $\{\Delta \vdash^{\Delta_{\mathsf{app}}; b} \grave{\tau}_i \rightsquigarrow \tau_i\ \mathsf{type}\}_{i \in L}$ — by assumption

(4) $\{\Delta \cup \Delta_{\mathsf{app}} \vdash \tau_i\ \mathsf{type}\}_{i \in L}$ — by IH over (3)

(5) $\Delta \cup \Delta_{\mathsf{app}} \vdash \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})\ \mathsf{type}$ — by Rule (B.1e) on (4)

Case (B.9f).

(1) $\grave{\tau} = \texttt{prsum}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L})$	by assumption
(2) $\tau = \texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L})$	by assumption
(3) $\{\Delta \vdash^{\Delta_{\text{app}}; b} \grave{\tau}_i \rightsquigarrow \tau_i \ \mathsf{type}\}_{i \in L}$	by assumption
(4) $\{\Delta \cup \Delta_{\text{app}} \vdash \tau_i \ \mathsf{type}\}_{i \in L}$	by IH over (3)
(5) $\Delta \cup \Delta_{\text{app}} \vdash \texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \ \mathsf{type}$	by Rule (B.1f) on (4)

Case (B.9g).

(1) $\grave{\tau} = \texttt{splicedt}[m; n]$	by assumption
(2) $\mathsf{parseUTyp}(\mathsf{subseq}(b; m; n)) = \hat{\tau}$	by assumption
(3) $\langle \mathcal{D}; \Delta_{\text{app}} \rangle \vdash \hat{\tau} \rightsquigarrow \tau \ \mathsf{type}$	by assumption
(4) $\Delta \cap \Delta_{\text{app}} = \emptyset$	by assumption
(5) $\Delta_{\text{app}} \vdash \tau \ \mathsf{type}$	by Lemma B.24 on (3)
(6) $\Delta \cup \Delta_{\text{app}} \vdash \tau \ \mathsf{type}$	by Lemma B.2 over Δ on (5) and exchange over Δ

□

B.4.2 Typed Pattern Expansion

Theorem B.26 (Typed Pattern Expansion).

1. *If* $\langle \mathcal{D}; \Delta \rangle \vdash_{\langle \mathcal{A}; \Phi \rangle} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \langle \mathcal{G}; \Gamma \rangle$ *then* $\Delta \vdash p : \tau \dashv\!\mid \Gamma$.
2. *If* $\grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\langle \mathcal{D}; \Delta \rangle; \langle \mathcal{A}; \Phi \rangle; b} \langle \mathcal{G}; \Gamma \rangle$ *then* $\Delta \vdash p : \tau \dashv\!\mid \Gamma$.

Proof. By mutual rule induction over Rules (B.8) and Rules (B.12).

1. We induct on the premise. In the following, let $\hat{\Delta} = \langle \mathcal{D}; \Delta \rangle$ and $\hat{\Gamma} = \langle \mathcal{G}; \Gamma \rangle$ and $\hat{\Phi} = \langle \mathcal{A}; \Phi \rangle$.

   Case (B.8a).

(1) $\hat{p} = \hat{x}$	by assumption
(2) $p = x$	by assumption
(3) $\Gamma = x : \tau$	by assumption
(4) $\Delta \vdash x : \tau \dashv\!\mid x : \tau$	by Rule (B.4a)

   Case (B.8b).

(1) $p = \texttt{wildp}$	by assumption
(2) $\Gamma = \emptyset$	by assumption
(3) $\Delta \vdash \texttt{wildp} : \tau \dashv\!\mid \emptyset$	by Rule (B.4b)

   Case (B.8c).

(1) $\hat{p} = \texttt{fold}(\hat{p}')$	by assumption
(2) $p = \texttt{foldp}(p')$	by assumption
(3) $\tau = \texttt{rec}(t.\tau')$	by assumption

(4) $\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p}' \rightsquigarrow p' : [\texttt{rec}(t.\tau')/t]\tau' \dashv\mid \hat{\Gamma}$ by assumption
(5) $\Delta \vdash p' : [\texttt{rec}(t.\tau')/t]\tau' \dashv\mid \Gamma$ by IH, part 1 on (4)
(6) $\Delta \vdash \texttt{foldp}(p') : \texttt{rec}(t.\tau') \dashv\mid \Gamma$ by Rule (B.4c) on (5)

Case (B.8d).

(1) $\hat{p} = \langle\{i \hookrightarrow \hat{p}_i\}_{i \in L}\rangle$ by assumption
(2) $p = \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L})$ by assumption
(3) $\tau = \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})$ by assumption
(4) $\{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p}_i \rightsquigarrow p_i : \tau_i \dashv\mid \langle\mathcal{G}_i; \Gamma_i\rangle\}_{i \in L}$ by assumption
(5) $\Gamma = \cup_{i \in L}\Gamma_i$ by assumption
(6) $\{\Delta \vdash p_i : \tau_i \dashv\mid \Gamma_i\}_{i \in L}$ by IH, part 1 over (4)
(7) $\Delta \vdash \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\mid \cup_{i \in L} \Gamma_i$
by Rule (B.4d) on (6)

Case (B.8e).

(1) $\hat{p} = \texttt{inj}[\ell](\hat{p}')$ by assumption
(2) $p = \texttt{injp}[\ell](p')$ by assumption
(3) $\tau = \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau')$ by assumption
(4) $\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p}' \rightsquigarrow p' : \tau' \dashv\mid \hat{\Gamma}$ by assumption
(5) $\Delta \vdash p' : \tau' \dashv\mid \Gamma$ by IH, part 1 on (4)
(6) $\Delta \vdash \texttt{injp}[\ell](p') : \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau') \dashv\mid \Gamma$
by Rule (B.4e) on (5)

Case (B.8f).

(1) $\hat{p} = \hat{a}$ ‘b‘ by assumption
(2) $\mathcal{A} = \mathcal{A}', \hat{a} \rightsquigarrow a$ by assumption
(3) $\Phi = \Phi', a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}})$ by assumption
(4) $b \downarrow_{\mathsf{Body}} e_{\text{body}}$ by assumption
(5) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}[\texttt{SuccessP}](e_{\text{proto}})$ by assumption
(6) $e_{\text{proto}} \uparrow_{\mathsf{PrPat}} \grave{p}$ by assumption
(7) $\grave{p} \rightsquigarrow p : \tau \dashv\mid^{\hat{\Delta}; \langle\mathcal{A};\Phi\rangle; b} \langle\mathcal{G}; \Gamma\rangle$ by assumption
(8) $\Delta \vdash p : \tau \dashv\mid \Gamma$ by IH, part 2 on (7)

2. We induct on the premise. In the following, let $\hat{\Gamma} = \langle\mathcal{G}; \Gamma\rangle$ and $\hat{\Delta} = \langle\mathcal{D}; \Delta\rangle$ and $\hat{\Phi} = \langle\mathcal{A}; \Phi\rangle$.

Case (B.12a).

(1) $p = \texttt{wildp}$ by assumption
(2) $\Gamma = \emptyset$ by assumption
(3) $\Delta \vdash \texttt{wildp} : \tau \dashv\mid \emptyset$ by Rule (B.4b)

Case (B.12b).

(1) $\grave{p} = \texttt{prfoldp}(\grave{p}')$ by assumption
(2) $p = \texttt{foldp}(p')$ by assumption
(3) $\tau = \texttt{rec}(t.\tau')$ by assumption

(4) $\grave{p}' \rightsquigarrow p' : [\mathtt{rec}(t.\tau')/t]\tau' \dashv\!\mid^{\hat{\Delta};\hat{\Phi};b} \hat{\Gamma}$ by assumption

(5) $\Delta \vdash p' : [\mathtt{rec}(t.\tau')/t]\tau' \dashv\!\mid \Gamma$ by IH, part 2 on (4)

(6) $\Delta \vdash \mathtt{foldp}(p') : \mathtt{rec}(t.\tau') \dashv\!\mid \Gamma$ by Rule (B.4c) on (5)

Case (B.12c).

(1) $\grave{p} = \mathtt{prtplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i \in L})$ by assumption

(2) $p = \mathtt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L})$ by assumption

(3) $\tau = \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})$ by assumption

(4) $\{\grave{p}_i \rightsquigarrow p_i : \tau_i \dashv\!\mid^{\hat{\Delta};\hat{\Phi};b} \langle \mathcal{G}_i; \Gamma_i \rangle\}_{i \in L}$ by assumption

(5) $\Gamma = \cup_{i \in L} \Gamma_i$ by assumption

(6) $\{\Delta \vdash p_i : \tau_i \dashv\!\mid \Gamma_i\}_{i \in L}$ by IH, part 2 over (4)

(7) $\Delta \vdash \mathtt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\!\mid \cup_{i \in L} \Gamma_i$
by Rule (B.4d) on (6)

Case (B.12d).

(1) $\grave{p} = \mathtt{prinjp}[\ell](\grave{p}')$ by assumption

(2) $p = \mathtt{injp}[\ell](p')$ by assumption

(3) $\tau = \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau')$ by assumption

(4) $\grave{p}' \rightsquigarrow p' : \tau' \dashv\!\mid^{\hat{\Delta};\hat{\Phi};b} \hat{\Gamma}$ by assumption

(5) $\Delta \vdash p' : \tau' \dashv\!\mid \Gamma$ by IH, part 2 on (4)

(6) $\Delta \vdash \mathtt{injp}[\ell](p') : \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau') \dashv\!\mid \Gamma$
by Rule (B.4e) on (5)

Case (B.12e).

(1) $\grave{p} = \mathtt{splicedp}[m; n; \grave{\tau}]$ by assumption

(2) $\emptyset \vdash^{\hat{\Delta};b} \grave{\tau} \rightsquigarrow \tau$ type by assumption

(3) $\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{p}$ by assumption

(4) $\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \hat{\Gamma}$ by assumption

(5) $\Delta \vdash p : \tau \dashv\!\mid \Gamma$ by IH, part 1 on (4)

The mutual induction can be shown to be well-founded by showing that the following numeric metric on the judgements that we induct on is decreasing:

$$\|\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \hat{\Gamma}\| = \|\hat{p}\|$$
$$\|\grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\hat{\Delta};\hat{\Phi};b} \hat{\Gamma}\| = \|b\|$$

where $\|b\|$ is the length of b and $\|\hat{p}\|$ is the sum of the lengths of the literal bodies in $\hat{p}$, as defined in Sec. B.2.1.

The only case in the proof of part 1 that invokes part 2 is Case (B.8f). There, we have that the metric remains stable:

$$\begin{aligned}&\|\hat{\Delta} \vdash_{\hat{\Phi}} \hat{a}\ \text{‘}b\text{‘} \rightsquigarrow p : \tau \dashv\!\vert\ \hat{\Gamma}\|\\ =&\|\grave{p} \rightsquigarrow p : \tau \dashv\!\vert^{\hat{\Delta};\hat{\Phi};b}\ \hat{\Gamma}\|\\ =&\|b\|\end{aligned}$$

The only case in the proof of part 2 that invokes part 1 is Case (B.12e). There, we have that $\mathsf{parseUPat}(\mathsf{subseq}(b;m;n)) = \hat{p}$ and the IH is applied to the judgement $\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\vert\ \hat{\Gamma}$. Because the metric is stable when passing from part 1 to part 2, we must have that it is strictly decreasing in the other direction:

$$\|\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\vert\ \hat{\Gamma}\| < \|\mathtt{splicedp}[m;n;\grave{\tau}] \rightsquigarrow p : \tau \dashv\!\vert^{\hat{\Delta};\hat{\Phi};b}\ \hat{\Gamma}\|$$

i.e. by the definitions above,

$$\|\hat{p}\| < \|b\|$$

This is established by appeal to Condition B.17, which states that subsequences of b are no longer than b, and the Condition B.13, which states that an unexpanded pattern constructed by parsing a textual sequence b is strictly smaller, as measured by the metric defined above, than the length of b, because some characters must necessarily be used to apply the pattern TSM and delimit each literal body. Combining Conditions B.17 and B.13, we have that $\|\hat{p}\| < \|b\|$ as needed. □

B.4.3 Typed Expression Expansion

Theorem B.27 (Typed Expansion (Strong))**.**

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

   (b) If $\langle \mathcal{D};\Delta\rangle\ \langle \mathcal{G};\Gamma\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \rightsquigarrow r : \tau \Rrightarrow \tau'$ then $\Delta\ \Gamma \vdash r : \tau \Rrightarrow \tau'$.

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

   (b) If $\Delta\ \Gamma \vdash^{\langle \mathcal{D};\Delta_{app}\rangle;\langle \mathcal{G};\Gamma_{app}\rangle;\hat{\Psi};\hat{\Phi};b} \grave{r} \rightsquigarrow r : \tau \Rrightarrow \tau'$ and $\Delta \cap \Delta_{app} = \emptyset$ and $dom(\Gamma) \cap dom(\Gamma_{app}) = \emptyset$ then $\Delta \cup \Delta_{app}\ \Gamma \cup \Gamma_{app} \vdash r : \tau \Rrightarrow \tau'$.

Proof. By mutual rule induction over Rules (B.6), Rule (B.7), Rules (B.10) and Rule (B.11).

1. In the following, let $\hat{\Delta} = \langle \mathcal{D};\Delta\rangle$ and $\hat{\Gamma} = \langle \mathcal{G};\Gamma\rangle$.

   (a) **Case** (B.6a)**.**

(1) $\hat{e} = \hat{x}$	by assumption
(2) $e = x$	by assumption
(3) $\Gamma = \Gamma', x : \tau$	by assumption
(4) $\Delta\ \Gamma', x : \tau \vdash x : \tau$	by Rule (B.2a)

   Case (B.6b)**.**

(1) $\hat{e} = \hat{e}' : \hat{\tau}$	by assumption

(2) $\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type}$ — by assumption
(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \rightsquigarrow e : \tau$ — by assumption
(4) $\Delta\ \Gamma \vdash e : \tau$ — by IH, part 1(a) on (3)

Case (B.6c).

(1) $\hat{e} = \mathtt{let\,val}\ \hat{x} = \hat{e}_1\ \mathtt{in}\ \hat{e}_2$ — by assumption
(2) $e = \mathtt{ap(lam\{}\tau_1\mathtt{\}(}x.e_2\mathtt{);}e_1\mathtt{)}$ — by assumption
(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_1 \rightsquigarrow e_1 : \tau_1$ — by assumption
(4) $\hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau_1 \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_2 \rightsquigarrow e_2 : \tau$ — by assumption
(5) $\Delta\ \Gamma \vdash e_1 : \tau_1$ — by IH, part 1(a) on (3)
(6) $\Delta\ \Gamma, x : \tau \vdash e_2 : \tau$ — by IH, part 1(a) on (4)
(7) $\Delta\ \Gamma \vdash \mathtt{lam\{}\tau_1\mathtt{\}(}x.e_2\mathtt{)} : \mathtt{parr(}\tau_1\mathtt{;}\tau\mathtt{)}$ — by Rule (B.2b) on (6)
(8) $\Delta\ \Gamma \vdash \mathtt{ap(lam\{}\tau_1\mathtt{\}(}x.e_2\mathtt{);}e_1\mathtt{)} : \tau$ — by Rule (B.2c) on (7) and (5)

Case (B.6d).

(1) $\hat{e} = \lambda \hat{x}{:}\hat{\tau}_1.\hat{e}'$ — by assumption
(2) $e = \mathtt{lam\{}\tau_1\mathtt{\}(}x.e'\mathtt{)}$ — by assumption
(3) $\tau = \mathtt{parr(}\tau_1\mathtt{;}\tau_2\mathtt{)}$ — by assumption
(4) $\hat{\Delta} \vdash \hat{\tau}_1 \rightsquigarrow \tau_1\ \mathsf{type}$ — by assumption
(5) $\hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau_1 \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \rightsquigarrow e' : \tau_2$ — by assumption
(6) $\Delta \vdash \tau_1\ \mathsf{type}$ — by Lemma B.24 on (4)
(7) $\Delta\ \Gamma, x : \tau_1 \vdash e' : \tau_2$ — by IH, part 1(a) on (5)
(8) $\Delta\ \Gamma \vdash \mathtt{lam\{}\tau_1\mathtt{\}(}x.e'\mathtt{)} : \mathtt{parr(}\tau_1\mathtt{;}\tau_2\mathtt{)}$ — by Rule (B.2b) on (6) and (7)

Case (B.6e).

(1) $\hat{e} = \hat{e}_1(\hat{e}_2)$ — by assumption
(2) $e = \mathtt{ap(}e_1\mathtt{;}e_2\mathtt{)}$ — by assumption
(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_1 \rightsquigarrow e_1 : \mathtt{parr(}\tau_2\mathtt{;}\tau\mathtt{)}$ — by assumption
(4) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_2 \rightsquigarrow e_2 : \tau_2$ — by assumption
(5) $\Delta\ \Gamma \vdash e_1 : \mathtt{parr(}\tau_2\mathtt{;}\tau\mathtt{)}$ — by IH, part 1(a) on (3)
(6) $\Delta\ \Gamma \vdash e_2 : \tau_2$ — by IH, part 1(a) on (4)
(7) $\Delta\ \Gamma \vdash \mathtt{ap(}e_1\mathtt{;}e_2\mathtt{)} : \tau$ — by Rule (B.2c) on (5) and (6)

Case (B.6f) **through** (B.6m). These cases follow analagously, i.e. we apply Lemma B.24 to or over the type expansion premises and the IH part 1(a) to or over the typed expression expansion premises and then apply the corresponding typing rule in Rules (B.2d) through (B.2k).

Case (B.6n).

(1) $\hat{e} = \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}'\ \texttt{by static}\ e_{\text{parse}}\ \texttt{in}\ \hat{e}'$ by assumption

(2) $\hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'\ \mathsf{type}$ by assumption

(3) $\emptyset\ \emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResultSE)}$ by assumption

(4) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{setsm}(\tau'; e_{\text{parse}}); \hat{\Phi}} \hat{e}' \rightsquigarrow e : \tau$ by assumption

(5) $\Delta \vdash \tau'\ \mathsf{type}$ by Lemma B.24 to (2)

(6) $\Delta\ \Gamma \vdash e : \tau$ by IH, part 1(a) on (4)

Case (B.6o).

(1) $\hat{e} = \hat{a}\ \texttt{`}b\texttt{`}$ by assumption

(2) $\mathcal{A} = \mathcal{A}', \hat{a} \rightsquigarrow a$ by assumption

(3) $\Psi = \Psi', a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}})$ by assumption

(4) $b \downarrow_{\mathsf{Body}} e_{\text{body}}$ by assumption

(5) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj[SuccessE]}(e_{\text{proto}})$ by assumption

(6) $e_{\text{proto}} \uparrow_{\mathsf{PrExpr}} \grave{e}$ by assumption

(7) $\emptyset\ \emptyset \vdash^{\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; \hat{\Phi}; b} \grave{e} \rightsquigarrow e : \tau$ by assumption

(8) $\emptyset \cap \Delta = \emptyset$ by finite set intersection

(9) $\emptyset \cap \text{dom}(\Gamma) = \emptyset$ by finite set intersection

(10) $\emptyset \cup \Delta\ \emptyset \cup \Gamma \vdash e : \tau$ by IH, part 2(a) on (7), (8), and (9)

(11) $\Delta\ \Gamma \vdash e : \tau$ by finite set and finite function identity over (10)

Case (B.6p).

(1) $\hat{e} = \texttt{match}\ \hat{e}'\ \{\hat{r}_i\}_{1 \le i \le n}$ by assumption

(2) $e = \texttt{match}[n](e'; \{r_i\}_{1 \le i \le n})$ by assumption

(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e}' \rightsquigarrow e' : \tau'$ by assumption

(4) $\{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{r}_i \rightsquigarrow r_i : \tau' \Mapsto \tau\}_{1 \le i \le n}$ by assumption

(5) $\Delta\ \Gamma \vdash e' : \tau'$ by IH, part 1(a) on (3)

(6) $\{\Delta\ \Gamma \vdash r_i : \tau' \Mapsto \tau\}_{1 \le i \le n}$ by IH, part 1(b) over (4)

(7) $\Delta\ \Gamma \vdash \texttt{match}[n](e'; \{r_i\}_{1 \le i \le n}) : \tau$ by Rule (B.2l) on (5) and (6)

Case (B.6q).

(1) $\hat{e} = \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}'\ \texttt{for patterns by static}\ e_{\text{parse}}\ \texttt{in}\ \hat{e}'$ by assumption

(2) $\hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'\ \mathsf{type}$ by assumption

(3) $\emptyset\ \emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResultSE)}$ by assumption

(4) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{sptsm}(\tau'; e_{\text{parse}})} \hat{e}' \rightsquigarrow e : \tau$ by assumption

(5) $\Delta \vdash \tau'$ type — by Lemma B.24 to (2)
(6) $\Delta\, \Gamma \vdash e : \tau$ — by IH, part 1(a) on (4)

(b) **Case** (B.7).

(1) $\hat{r} = \hat{p} \Rightarrow \hat{e}$ — by assumption
(2) $r = \texttt{rule}(p.e)$ — by assumption
(3) $\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\!\vert \langle \mathcal{A}'; \Gamma \rangle$ — by assumption
(4) $\hat{\Delta}\, \langle \mathcal{A} \uplus \mathcal{A}'; \Gamma \cup \Gamma \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e : \tau'$ — by assumption
(5) $\Delta \vdash p : \tau \dashv\!\!\vert \Gamma$ — by Theorem B.26, part 1 on (3)
(6) $\Delta\, \Gamma \cup \Gamma \vdash e : \tau'$ — by IH, part 1(a) on (4)
(7) $\Delta\, \Gamma \vdash \texttt{rule}(p.e) : \tau \Mapsto \tau'$ — by Rule (B.3) on (5) and (6)

2. In the following, let $\hat{\Delta} = \langle \mathcal{D}; \Delta_{\text{app}} \rangle$ and $\hat{\Gamma} = \langle \mathcal{G}; \Gamma_{\text{app}} \rangle$.

(a) **Case** (B.10a).

(1) $\grave{e} = x$ — by assumption
(2) $e = x$ — by assumption
(3) $\Gamma = \Gamma', x : \tau$ — by assumption
(4) $\Delta \cup \Delta_{\text{app}}\, \Gamma', x : \tau \vdash x : \tau$ — by Rule (B.2a)
(5) $\Delta \cup \Delta_{\text{app}}\, \Gamma', x : \tau \cup \Gamma_{\text{app}} \vdash x : \tau$ — by Lemma B.2 over Γ_{app} to (4)

Case (B.10d).

(1) $\grave{e} = \texttt{prlam}\{\grave{\tau}_1\}(x.\grave{e}')$ — by assumption
(2) $e = \texttt{lam}\{\tau_1\}(x.e')$ — by assumption
(3) $\tau = \texttt{parr}(\tau_1; \tau_2)$ — by assumption
(4) $\Delta \vdash^{\hat{\Delta}_{\text{app}}; b} \grave{\tau}_1 \rightsquigarrow \tau_1$ type — by assumption
(5) $\Delta\, \Gamma, x : \tau_1 \vdash^{\hat{\Delta}_{\text{app}}; \hat{\Gamma}_{\text{app}}; \hat{\Psi}; \hat{\Phi}; b} \grave{e}' \rightsquigarrow e' : \tau_2$ — by assumption
(6) $\Delta \cap \Delta_{\text{app}} = \emptyset$ — by assumption
(7) $\text{dom}(\Gamma) \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset$ — by assumption
(8) $x \notin \text{dom}(\Gamma_{\text{app}})$ — by identification convention
(9) $\text{dom}(\Gamma, x : \tau_1) \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset$ — by (7) and (8)
(10) $\Delta \cup \Delta_{\text{app}} \vdash \tau_1$ type — by Lemma B.25 on (4) and (6)
(11) $\Delta \cup \Delta_{\text{app}}\, \Gamma, x : \tau_1 \cup \Gamma_{\text{app}} \vdash e' : \tau_2$ — by IH, part 2(a) on (5), (6) and (9)
(12) $\Delta \cup \Delta_{\text{app}}\, \Gamma \cup \Gamma_{\text{app}}, x : \tau_1 \vdash e' : \tau_2$ — by exchange over Γ_{app} on (11)
(13) $\Delta \cup \Delta_{\text{app}}\, \Gamma \cup \Gamma_{\text{app}} \vdash \texttt{lam}\{\tau_1\}(x.e') : \texttt{parr}(\tau_1; \tau_2)$ — by Rule (B.2b) on (10) and (12)

Case (B.10e).

(1) $\grave{e} = \texttt{prap}(\grave{e}_1; \grave{e}_2)$	by assumption
(2) $e = \texttt{ap}(e_1; e_2)$	by assumption
(3) $\Delta\ \Gamma \vdash^{\hat{\Delta}_{\text{app}}; \hat{\Gamma}_{\text{app}}; \hat{\Psi}; \hat{\Phi}; b} \grave{e}_1 \leadsto e_1 : \texttt{parr}(\tau_2; \tau)$	by assumption
(4) $\Delta\ \Gamma \vdash^{\hat{\Delta}_{\text{app}}; \hat{\Gamma}_{\text{app}}; \hat{\Psi}; \hat{\Phi}; b} \grave{e}_2 \leadsto e_2 : \tau_2$	by assumption
(5) $\Delta \cap \Delta_{\text{app}} = \emptyset$	by assumption
(6) $\text{dom}(\Gamma) \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset$	by assumption
(7) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash e_1 : \texttt{parr}(\tau_2; \tau)$	by IH, part 2(a) on (3), (5) and (6)
(8) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash e_2 : \tau_2$	by IH, part 2(a) on (4), (5) and (6)
(9) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash \texttt{ap}(e_1; e_2) : \tau$	by Rule (B.2c) on (7) and (8)

Case (B.10f).

(1) $\grave{e} = \texttt{prtlam}(t.\grave{e}')$	by assumption
(2) $e = \texttt{tlam}(t.e')$	by assumption
(3) $\tau = \texttt{all}(t.\tau')$	by assumption
(4) $\Delta, t\ \mathsf{type}\ \Gamma \vdash^{\hat{\Delta}_{\text{app}}; \hat{\Gamma}_{\text{app}}; \hat{\Psi}; \hat{\Phi}; b} \grave{e}' \leadsto e' : \tau'$	by assumption
(5) $\Delta \cap \Delta_{\text{app}} = \emptyset$	by assumption
(6) $\text{dom}(\Gamma) \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset$	by assumption
(7) $t\ \mathsf{type} \notin \Delta_{\text{app}}$	by identification convention
(8) $\Delta, t\ \mathsf{type} \cap \Delta_{\text{app}} = \emptyset$	by (5) and (7)
(9) $\Delta, t\ \mathsf{type} \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash e' : \tau'$	by IH, part 2(a) on (4), (8) and (6)
(10) $\Delta \cup \Delta_{\text{app}}, t\ \mathsf{type}\ \Gamma \cup \Gamma_{\text{app}} \vdash e' : \tau'$	by exchange over Δ_{app} on (9)
(11) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash \texttt{tlam}(t.e') : \texttt{all}(t.\tau')$	by Rule (B.2d) on (10)

Case (B.10g) **through** (B.10m). These cases follow analagously, i.e. we apply the IH, part 2(a) to all proto-expression validation judgements, Lemma B.25 to all proto-type validation judgements, the identification convention to ensure that extended contexts remain disjoint, weakening and exchange as needed, and the corresponding typing rule in Rules (B.2e) through (B.2k).

Case (B.10n).

(1) $\grave{e} = \texttt{splicede}[m; n; \grave{\tau}]$	by assumption
(2) $\mathbb{E} = \langle \mathcal{D}; \Delta_{\text{app}} \rangle;\ \langle \mathcal{G}; \Gamma_{\text{app}} \rangle;\ \hat{\Psi};\ b$	by assumption
(3) $\emptyset \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau} \leadsto \tau\ \mathsf{type}$	by assumption
(4) $\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e}$	by assumption

(5) $\hat{\Delta}_{\text{app}}\ \hat{\Gamma}_{\text{app}} \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau$ — by assumption
(6) $\Delta \cap \Delta_{\text{app}} = \emptyset$ — by assumption
(7) $\text{dom}(\Gamma) \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset$ — by assumption
(8) $\Delta_{\text{app}}\ \Gamma_{\text{app}} \vdash e : \tau$ — by IH, part 1 on (5)
(9) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash e : \tau$ — by Lemma B.2 over Δ and Γ and exchange on (8)

Case (B.10o).

(1) $\grave{e} = \texttt{prmatch}[n](\grave{e}'; \{\grave{r}_i\}_{1 \le i \le n})$ — by assumption
(2) $e = \texttt{match}[n](e'; \{r_i\}_{1 \le i \le n})$ — by assumption
(3) $\Delta\ \Gamma \vdash^{\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; \hat{\Phi}; b} \grave{e}' \rightsquigarrow e' : \tau'$ — by assumption
(4) $\{\Delta\ \Gamma \vdash^{\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; \hat{\Phi}; b} \grave{r}_i \rightsquigarrow r_i : \tau' \Rrightarrow \tau\}_{1 \le i \le n}$ — by assumption
(5) $\Delta \cap \Delta_{\text{app}} = \emptyset$ — by assumption
(6) $\text{dom}(\Gamma) \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset$ — by assumption
(7) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash e' : \tau'$ — by IH, part 2(a) on (3), (5) and (6)
(8) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash r : \tau' \Rrightarrow \tau$ — by IH, part 2(b) on (4), (5) and (6)
(9) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash \texttt{match}[n](e'; \{r_i\}_{1 \le i \le n}) : \tau$ — by Rule (B.21) on (7) and (8)

(b) There is only one case.

Case (B.11).

(1) $\grave{r} = \texttt{prrule}(p.\grave{e})$ — by assumption
(2) $r = \texttt{rule}(p.e)$ — by assumption
(3) $\Delta \vdash p : \tau \dashv\!\mid \Gamma'$ — by assumption
(4) $\Delta\ \Gamma \cup \Gamma' \vdash^{\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; \hat{\Phi}; b} \grave{e} \rightsquigarrow e : \tau'$ — by assumption
(5) $\Delta \cap \Delta_{\text{app}} = \emptyset$ — by assumption
(6) $\text{dom}(\Gamma) \cap \text{dom}(\Gamma') = \emptyset$ — by identification convention
(7) $\text{dom}(\Gamma_{\text{app}}) \cap \text{dom}(\Gamma') = \emptyset$ — by identification convention
(8) $\text{dom}(\Gamma) \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset$ — by assumption
(9) $\text{dom}(\Gamma \cup \Gamma') \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset$ — by standard finite set definitions and identities on (6), (7) and (8)
(10) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma' \cup \Gamma_{\text{app}} \vdash e : \tau'$ — by IH, part 2(a) on (4), (5) and (9)
(11) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \cup \Gamma' \vdash e : \tau'$ — by exchange of Γ' and Γ_{app} on (10)

(12) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash \texttt{rule}(p.e) : \tau \Rightarrow \tau'$ by Rule (B.3) on (3) and (11)

The mutual induction can be shown to be well-founded by showing that the following numeric metric on the judgements that we induct on is decreasing:

$$\|\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau\| = \|\hat{e}\|$$
$$\|\Delta\ \Gamma \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e : \tau\| = \|b\|$$

where $\|b\|$ is the length of b and $\|\hat{e}\|$ is the sum of the lengths of the seTSM literal bodies in $\hat{e}$, as defined in Sec. B.2.1.

The only case in the proof of part 1 that invokes part 2 is Case (B.6o). There, we have that the metric remains stable:

$$\begin{aligned}&\|\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{a}\ \text{`}b\text{`} \rightsquigarrow e : \tau\|\\ =&\|\emptyset\ \emptyset \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e : \tau\|\\ =&\|b\|\end{aligned}$$

The only case in the proof of part 2 that invokes part 1 is Case (B.10n). There, we have that $\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e}$ and the IH is applied to the judgement $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau$. Because the metric is stable when passing from part 1 to part 2, we must have that it is strictly decreasing in the other direction:

$$\|\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau\| < \|\Delta\ \Gamma \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \texttt{splicede}[m; n; \grave{\tau}] \rightsquigarrow e : \tau\|$$

i.e. by the definitions above,

$$\|\hat{e}\| < \|b\|$$

This is established by appeal to Condition B.17, which states that subsequences of b are no longer than b, and Condition B.12, which states that an unexpanded expression constructed by parsing a textual sequence b is strictly smaller, as measured by the metric defined above, than the length of b, because some characters must necessarily be used to apply a TSM and delimit each literal body. Combining these conditions, we have that $\|\hat{e}\| < \|b\|$ as needed. □

Theorem B.28 (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* $\Delta\ \Gamma \vdash e : \tau$*.*

Proof. This theorem follows immediately from Theorem B.27, part 1(a). □

B.4.4 Abstract Reasoning Principles

Lemma B.29 (Proto-Type Expansion Decomposition)**.** *If* $\Delta \vdash^{\langle \mathcal{D}; \Delta_{app}\rangle; b} \grave{\tau} \rightsquigarrow \tau\ \mathsf{type}$ *where* $\mathsf{summary}(\grave{\tau}) = \{\texttt{splicedt}[m_i; n_i]\}_{0 \le i < n}$ *then all of the following hold:*

1. $\{\langle \mathcal{D}; \Delta_{app}\rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow \tau_i\ \mathsf{type}\}_{0 \le i < n}$

2. $\tau = [\{\tau_i/t_i\}_{0\leq i<n}]\tau'$ *for some* τ' *and fresh* $\{t_i\}_{0\leq i<n}$ *(i.e.* $\{t_i \notin dom(\Delta)\}_{0\leq i<n}$ *and* $\{t_i \notin dom(\Delta_{app})\}_{0\leq i<n}$*)*
3. $\mathsf{fv}(\tau') \subset dom(\Delta) \cup \{t_i\}_{0\leq i<n}$

Proof. By rule induction over Rules (B.9). In the following, let $\hat{\Delta} = \langle \mathcal{D}; \Delta_{\text{app}} \rangle$ and $\mathbb{T} = \hat{\Delta}; b$.

Case (B.9a).

(1)	$\grave{\tau} = t$	by assumption
(2)	$\tau = t$	by assumption
(3)	$\Delta = \Delta', t\ \mathsf{type}$	by assumption
(4)	$\mathsf{summary}(\grave{\tau}) = \emptyset$	by definition
(5)	$\mathsf{fv}(t) = \{t\}$	by definition
(6)	$\{t\} \subset \text{dom}(\Delta) \cup \emptyset$	by definition

The conclusions hold as follows:

1. This conclusion holds trivially because $n = 0$.
2. Choose $\tau' = t$ and $\emptyset$.
3. (6)

Case (B.9b).

(1)	$\grave{\tau} = \mathtt{prparr}(\grave{\tau}_1; \grave{\tau}_2)$	by assumption
(2)	$\tau = \mathtt{parr}(\tau_1'; \tau_2')$	by assumption
(3)	$\Delta \vdash^{\mathbb{T}} \grave{\tau}_1 \rightsquigarrow \tau_1'\ \mathsf{type}$	by assumption
(4)	$\Delta \vdash^{\mathbb{T}} \grave{\tau}_2 \rightsquigarrow \tau_2'\ \mathsf{type}$	by assumption
(5)	$\mathsf{summary}(\grave{\tau}) = \mathsf{summary}(\grave{\tau}_1) \cup \mathsf{summary}(\grave{\tau}_2)$	by definition
(6)	$\mathsf{summary}(\grave{\tau}_1) = \{\mathtt{splicedt}[m_i; n_i]\}_{0\leq i<n'}$	by definition
(7)	$\mathsf{summary}(\grave{\tau}_2) = \{\mathtt{splicedt}[m_i; n_i]\}_{n'\leq i<n}$	by definition
(8)	$\{\langle \mathcal{D}; \Delta_{\text{app}} \rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow \tau_i\ \mathsf{type}\}_{0\leq i<n'}$	by IH on (3) and (6)
(9)	$\tau_1' = [\{\tau_i/t_i\}_{0\leq i<n'}]\tau_1''$ for some τ_1'' and fresh $\{t_i\}_{0\leq i<n'}$	by IH on (3) and (6)
(10)	$\mathsf{fv}(\tau_1'') \subset \text{dom}(\Delta) \cup \{t_i\}_{0\leq i<n'}$	by IH on (3) and (6)
(11)	$\{\langle \mathcal{D}; \Delta_{\text{app}} \rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow \tau_i\ \mathsf{type}\}_{n'\leq i<n}$	by IH on (4) and (7)
(12)	$\tau_2' = [\{\tau_i/t_i\}_{n'\leq i<n}]\tau_2''$ for some τ_2'' and fresh $\{t_i\}_{n'\leq i<n}$	by IH on (4) and (7)
(13)	$\mathsf{fv}(\tau_2'') \subset \text{dom}(\Delta) \cup \{t_i\}_{n'\leq i<n}$	by IH on (4) and (7)
(14)	$\{t_i\}_{0\leq i<n'} \cap \{t_i\}_{n'\leq i<n} = \emptyset$	by identification convention
(15)	$\mathsf{fv}(\tau_1'') \subset \text{dom}(\Delta) \cup \{t_i\}_{0\leq i<n}$	by (10) and (14)
(16)	$\mathsf{fv}(\tau_2'') \subset \text{dom}(\Delta) \cup \{t_i\}_{0\leq i<n}$	by (13) and (14)
(17)	$\tau_1' = [\{\tau_i/t_i\}_{0\leq i<n}]\tau_1''$	by substitution properties and (9) and (14)

(18) $\tau_2' = [\{\tau_i/t_i\}_{0\leq i<n}]\tau_2''$ by substitution properties and (12) and (14)

(19) $\texttt{parr}(\tau_1';\tau_2') = [\{\tau_i/t_i\}_{0\leq i<n}]\texttt{parr}(\tau_1'';\tau_2'')$ by substitution and (17) and (18)

(20) $\mathsf{fv}(\texttt{parr}(\tau_1'';\tau_2'')) = \mathsf{fv}(\tau_1'') \cup \mathsf{fv}(\tau_2'')$ by definition

(21) $\mathsf{fv}(\texttt{parr}(\tau_1'';\tau_2'')) \subset \mathrm{dom}(\Delta) \cup \{t_i\}_{0\leq i<n}$ by (20) and (15) and (16)

The conclusions hold as follows:

1. (8) $\cup$ (11)
2. Choosing $\{t_i\}_{0\leq i<n}$ and $\texttt{parr}(\tau_1'';\tau_2'')$, by (19)
3. (21)

Case (B.9c) **through** (B.9f). These cases follow by analagous inductive argument.

Case (B.9g).

(1) $\grave{\tau} = \texttt{splicedt}[m;n]$ by assumption

(2) $\mathsf{summary}(\texttt{splicedt}[m;n]) = \{\texttt{splicedt}[m;n]\}$ by definition

(3) $\mathsf{parseUTyp}(\mathsf{subseq}(b;m;n)) = \hat{\tau}$ by assumption

(4) $\langle\mathcal{D};\Delta_{\text{app}}\rangle \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type}$ by assumption

(5) $t \notin \mathrm{dom}(\Delta)$ by identification convention

(6) $t \notin \mathrm{dom}(\Delta_{\text{app}})$ by identification

(7) $\tau = [\tau/t]\tau$ by definition

(8) $\mathsf{fv}(t) \subset \Delta \cup \{t\}$ by definition

The conclusions hold as follows:

1. (3) and (4)
2. Choosing $\{t\}$ and t, by (5), (6) and (7)
3. (8)

$\square$

Lemma B.30 (Proto-Expression and Proto-Rule Expansion Decomposition).

1. *If* $\Delta\ \Gamma \vdash^{\langle\mathcal{D};\Delta_{app}\rangle;\langle\mathcal{G};\Gamma_{app}\rangle;\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e : \tau$ *where* $\mathsf{summary}(\grave{e}) = \{\texttt{splicedt}[m_i';n_i']\}_{0\leq i<n_{ty}} \cup \{\texttt{splicede}[m_i;n_i;\grave{\tau}_i]\}_{0\leq i<n_{exp}}$ *then all of the following hold:*
   - *(a)* $\{\langle\mathcal{D};\Delta_{app}\rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b;m_i';n_i')) \rightsquigarrow \tau_i'\ \mathsf{type}\}_{0\leq i<n_{ty}}$
   - *(b)* $\{\emptyset \vdash^{\langle\mathcal{D};\Delta_{app}\rangle;b} \grave{\tau}_i \rightsquigarrow \tau_i\ \mathsf{type}\}_{0\leq i<n_{exp}}$
   - *(c)* $\{\langle\mathcal{D};\Delta_{app}\rangle\ \langle\mathcal{G};\Gamma_{app}\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \mathsf{parseUExp}(\mathsf{subseq}(b;m_i;n_i)) \rightsquigarrow e_i : \tau_i\}_{0\leq i<n_{exp}}$
   - *(d)* $e = [\{\tau_i'/t_i\}_{0\leq i<n_{ty}}, \{e_i/x_i\}_{0\leq i<n_{exp}}]e'$ *for some* e' *and* $\{t_i\}_{0\leq i<n_{ty}}$ *and* $\{x_i\}_{0\leq i<n_{exp}}$ *such that* $\{t_i\}_{0\leq i<n_{ty}}$ *fresh (i.e.* $\{t_i \notin dom(\Delta)\}_{0\leq i<n_{ty}}$ *and* $\{t_i \notin dom(\Delta_{app})\}_{0\leq i<n_{ty}}$*) and* $\{x_i\}_{0\leq i<n_{exp}}$ *fresh (i.e.* $\{x_i \notin dom(\Gamma)\}_{0\leq i<n_{exp}}$ *and* $\{x_i \notin dom(\Gamma_{app})\}_{0\leq i<n_{ty}}$*)*
   - *(e)* $\mathsf{fv}(e') \subset dom(\Delta) \cup dom(\Gamma) \cup \{t_i\}_{0\leq i<n_{ty}} \cup \{x_i\}_{0\leq i<n_{exp}}$

2. *If* $\Delta\ \Gamma \vdash^{\langle \mathcal{D};\Delta_{app}\rangle;\langle \mathcal{G};\Gamma_{app}\rangle;\hat{\Psi};\hat{\Phi};b} \grave{r} \leadsto r : \tau \Mapsto \tau'$ *and*

$$\mathsf{summary}(\grave{r}) = \{\mathtt{splicedt}[m'_i;n'_i]\}_{0\leq i<n_{ty}} \cup \{\mathtt{splicede}[m_i;n_i;\grave{\tau}_i]\}_{0\leq i<n_{exp}}$$

then all of the following hold:

(a) $\{\langle \mathcal{D};\Delta_{app}\rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b;m'_i;n'_i)) \leadsto \tau'_i\ \mathsf{type}\}_{0\leq i<n_{ty}}$

(b) $\{\emptyset \vdash^{\langle \mathcal{D};\Delta_{app}\rangle;b} \grave{\tau}_i \leadsto \tau_i\ \mathsf{type}\}_{0\leq i<n_{exp}}$

(c) $\{\langle \mathcal{D};\Delta_{app}\rangle\ \langle \mathcal{G};\Gamma_{app}\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \mathsf{parseUExp}(\mathsf{subseq}(b;m_i;n_i)) \leadsto e_i : \tau_i\}_{0\leq i<n_{exp}}$

(d) $r = [\{\tau'_i/t_i\}_{0\leq i<n_{ty}}, \{e_i/x_i\}_{0\leq i<n_{exp}}]r'$ *for some* e' *and fresh* $\{t_i\}_{0\leq i<n_{ty}}$ *and fresh* $\{x_i\}_{0\leq i<n_{exp}}$

(e) $\mathsf{fv}(r') \subset dom(\Delta) \cup dom(\Gamma) \cup \{t_i\}_{0\leq i<n_{ty}} \cup \{x_i\}_{0\leq i<n_{exp}}$

Proof. By rule induction over Rules (B.10) and Rule (B.11). In the following, let $\hat{\Delta} = \langle \mathcal{D};\Delta_{\text{app}}\rangle$ and $\hat{\Gamma} = \langle \mathcal{G};\Gamma_{\text{app}}\rangle$ and $\mathbb{E} = \hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b$.

1. **Case** (B.10a).

(1) $\grave{e} = x$	by assumption
(2) $e = x$	by assumption
(3) $\Gamma = \Gamma', x : \tau$	by assumption
(4) $\mathsf{summary}(x) = \{\}$	by definition
(5) $\mathsf{fv}(x) = \{x\}$	by definition
(6) $\mathsf{fv}(x) \subset \text{dom}(\Gamma)$	by definition
(7) $\mathsf{fv}(x) \subset \text{dom}(\Gamma) \cup \text{dom}(\Delta)$	by (6) and definition of subset

The conclusions hold as follows:

(a) This conclusion holds trivially because $n_{\text{ty}} = 0$.

(b) This conclusion holds trivially because $n_{\text{exp}} = 0$.

(c) This conclusion holds trivially because $n_{\text{exp}} = 0$.

(d) Choose x, $\emptyset$ and $\emptyset$.

(e) (7)

Case (B.10b) **through** (B.10m). These cases follow by straightforward inductive argument.

Case (B.10n).

(1) $\grave{e} = \mathtt{splicede}[m;n;\grave{\tau}]$	by assumption
(2) $\mathsf{summary}(\mathtt{splicede}[m;n;\grave{\tau}]) = \mathsf{summary}(\grave{\tau}) \cup \{\mathtt{splicede}[m;n;\grave{\tau}]\}$	by definition
(3) $\mathsf{summary}(\grave{\tau}) = \{\mathtt{splicedt}[m'_i;n'_i]\}_{0\leq i<n_{\text{ty}}}$	by definition
(4) $\emptyset \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau} \leadsto \tau\ \mathsf{type}$	by assumption
(5) $\mathsf{parseUExp}(\mathsf{subseq}(b;m;n)) = \hat{e}$	by assumption
(6) $\langle \mathcal{D};\Delta_{\text{app}}\rangle\ \langle \mathcal{G};\Gamma_{\text{app}}\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e : \tau$	by assumption
(7) $\{\langle \mathcal{D};\Delta_{\text{app}}\rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b;m'_i;n'_i)) \leadsto \tau'_i\ \mathsf{type}\}_{0\leq i<n_{\text{ty}}}$	by Lemma B.29 on (4) and (3)

(8) $x \notin \text{dom}(\Gamma)$ by identification convention

(9) $x \notin \text{dom}(\Gamma_{\text{app}})$ by identification convention

(10) $x \notin \text{dom}(\Delta)$ by identificaiton convention

(11) $x \notin \text{dom}(\Delta_{\text{app}})$ by identification convention

(12) $e = [\{\tau_i'/t_i\}_{0 \le i < n_{\text{ty}}}, e/x]x$ by definition

(13) $\text{fv}(x) = \{x\}$ by definition

(14) $\text{fv}(x) \subset \text{dom}(\Delta) \cup \text{dom}(\Gamma) \cup \{t_i\}_{0 \le i < n_{\text{ty}}} \cup \{x\}$ by definition

The conclusions hold as follows:

(a) (7)

(b) $\{$(4)$\}$

(c) $\{$(6)$\}$

(d) Choosing x, $\{t_i\}_{0 \le i < n_{\text{ty}}}$ and $\{x\}$, by (8), (9), (10), (11) and (12).

(e) (14)

Case (B.10o).

(1) $\grave{e} = \texttt{prmatch}[n](\grave{e}'; \{\grave{r}_i\}_{1 \le i \le n})$ by assumption

(2) $e = \texttt{match}[n](\tau; e')\{r_i\}_{1 \le i \le n}$ by assumption

(3) $\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau'$ by assumption

(4) $\{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r}_j \rightsquigarrow r_j : \tau' \Rrightarrow \tau\}_{1 \le j \le n}$ by assumption

(5) $\mathsf{summary}(\texttt{prmatch}[n](\grave{e}'; \{\grave{r}_i\}_{1 \le i \le n})) = \mathsf{summary}(\grave{e}) \cup \bigcup_{0 \le i < n} \mathsf{summary}(\grave{r}_i)$ by definition

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

(7) $\{\mathsf{summary}(\grave{r}_j) = \{\texttt{splicedt}[m_{i,j}'; n_{i,j}']\}_{0 \le i < n_{\text{ty},j}} \cup \{\texttt{splicede}[m_{i,j}; n_{i,j}; \grave{\tau}_{i,j}]\}_{0 \le i < n_{\text{exp},j}}\}_{0 \le j < n}$ by definition

(8) $\{\langle \mathcal{D}; \Delta_{\text{app}} \rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m_i'; n_i')) \rightsquigarrow \tau_i'\ \mathsf{type}\}_{0 \le i < n_{\text{ty}}'}$ by IH, part 1 on (3) and (6)

(9) $\{\emptyset \vdash^{\langle \mathcal{D}; \Delta_{\text{app}} \rangle; b} \grave{\tau}_i \rightsquigarrow \tau_i\ \mathsf{type}\}_{0 \le i < n_{\text{exp}}'}$ by IH, part 1 on (3) and (6)

(10) $\{\langle \mathcal{D}; \Delta_{\text{app}} \rangle\ \langle \mathcal{G}; \Gamma_{\text{app}} \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \mathsf{parseUExp}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow e_i : \tau_i\}_{0 \le i < n_{\text{exp}}'}$ by IH, part 1 on (3) and (6)

(11) $e' = [\{\tau_i'/t_i\}_{0 \le i < n_{\text{ty}}'}, \{e_i/x_i\}_{0 \le i < n_{\text{exp}}'}]e''$ for some e'' and fresh $\{t_i\}_{0 \le i < n_{\text{ty}}'}$ and fresh $\{x_i\}_{0 \le i < n_{\text{exp}}'}$ by IH, part 1 on (3) and (6)

(12) $\mathsf{fv}(e'') \subset \mathsf{dom}(\Delta) \cup \mathsf{dom}(\Gamma) \cup \{t_i\}_{0 \le i < n'_{\mathsf{ty}}} \cup \{x_i\}_{0 \le i < n'_{\mathsf{exp}}}$ by IH, part 1 on (3) and (6)

(13) $\{\{\langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_{i,j}; n'_{i,j})) \rightsquigarrow \tau'_{i,j}\ \mathsf{type}\}_{0 \le i < n_{\mathsf{ty},j}}\}_{0 \le j < n}$ by IH, part 2 over (4) and (7)

(14) $\{\{\emptyset \vdash^{\langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle; b} \hat{\tau}_{i,j} \rightsquigarrow \tau_{i,j}\ \mathsf{type}\}_{0 \le i < n_{\mathsf{exp},j}}\}_{0 \le j < n}$ by IH, part 2 over (4) and (7)

(15) $\{\{\langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle\ \langle \mathcal{G}; \Gamma_{\mathsf{app}} \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \mathsf{parseUExp}(\mathsf{subseq}(b; m_{i,j}; n_{i,j})) \rightsquigarrow e_{i,j} : \tau_{i,j}\}_{0 \le i < n_{\mathsf{exp},j}}\}_{0 \le j < n}$ by IH, part 2 over (4) and (7)

(16) $\{r_j = [\{\tau'_{i,j}/t_{i,j}\}_{0 \le i < n_{\mathsf{ty},j}}, \{e_{i,j}/x_{i,j}\}_{0 \le i < n_{\mathsf{exp},j}}]r'_j\}_{0 \le j < n}$ for some $\{r'_j\}_{0 \le j < n}$ and fresh $\{\{t_{i,j}\}_{0 \le i < n_{\mathsf{ty},j}}\}_{0 \le j < n}$ and fresh $\{\{x_{i,j}\}_{0 \le i < n_{\mathsf{exp},j}}\}_{0 \le j < n}$ by IH, part 2 over (4) and (7)

(17) $\{\mathsf{fv}(r'_j) \subset \mathsf{dom}(\Delta) \cup \mathsf{dom}(\Gamma) \cup \{t_{i,j}\}_{0 \le i < n_{\mathsf{ty},j}} \cup \{x_{i,j}\}_{0 \le i < n_{\mathsf{exp},j}}\}_{0 \le j < n}$ by IH, part 2 over (4) and (7)

(18) $(\cup_{0 \le j < n} \{t_{i,j}\}_{0 \le i < n_{\mathsf{ty},j}}) \cap \{t_i\}_{0 \le i < n'_{\mathsf{ty}}} = \emptyset$ by identification convention

(19) $(\cup_{0 \le j < n} \{x_{i,j}\}_{0 \le i < n_{\mathsf{exp},j}}) \cap \{x_i\}_{0 \le i < n'_{\mathsf{exp}}} = \emptyset$ by identification convention

(20) $e' = [\{\tau'_i/t_i\}_{0 \le i < n'_{\mathsf{ty}}} \cup_{0 \le j < n} \{\tau_{i,j}/t_{i,j}\}_{0 \le i < n_{\mathsf{ty},j}}, \{e_i/x_i\}_{0 \le i < n_{\mathsf{exp}}} \cup_{0 \le j < n} \{\tau_{i,j}/t_{i,j}\}_{0 \le i < n_{\mathsf{ty},j}}]e''$ by substitution properties and (11) and (12) and (18) and (19)

(21) $\{r_j = [\{\tau'_i/t_i\}_{0 \le i < n'_{\mathsf{ty}}} \cup_{0 \le j < n} \{\tau_{i,j}/t_{i,j}\}_{0 \le i < n_{\mathsf{ty},j}}, \{e_i/x_i\}_{0 \le i < n_{\mathsf{exp}}} \cup_{0 \le j < n} \{\tau_{i,j}/t_{i,j}\}_{0 \le i < n_{\mathsf{ty},j}}]r'_j\}_{0 \le j < n}$ by substitution properties and (16) and (17) and (18) and (19)

(22) $e = [\{\tau'_i/t_i\}_{0 \le i < n'_{\mathsf{ty}}} \cup_{0 \le j < n} \{\tau_{i,j}/t_{i,j}\}_{0 \le i < n_{\mathsf{ty},j}}, \{e_i/x_i\}_{0 \le i < n'_{\mathsf{exp}}} \cup_{0 \le j < n} \{e_{i,j}/x_{i,j}\}_{0 \le i < n_{\mathsf{exp},j}}]\mathtt{match}[n](e''; \{r'_i\}_{1 \le i \le n})$ by (20) and (21) and definition of substitution

(23) $\mathsf{fv}(e'') \subset \mathsf{dom}(\Delta) \cup \mathsf{dom}(\Gamma) \cup \{t_i\}_{0 \le i < n'_{\mathsf{ty}}} \cup_{0 \le j < n} \{t_{i,j}\}_{0 \le i < n_{\mathsf{ty},j}} \cup \{x_i\}_{0 \le i < n'_{\mathsf{exp}}} \cup_{0 \le j < n} \{x_{i,j}\}_{0 \le i < n_{\mathsf{exp},j}}$ by (12) and (18) and (19)

(24) $\{\mathsf{fv}(r'_j) \subset \mathsf{dom}(\Delta) \cup \mathsf{dom}(\Gamma) \cup \{t_i\}_{0 \le i < n'_{\mathsf{ty}}} \cup_{0 \le j < n} \{t_{i,j}\}_{0 \le i < n_{\mathsf{ty},j}} \cup$

$\{x_i\}_{0\le i<n'_{\text{exp}}} \cup_{0\le j<n} \{x_{i,j}\}_{0\le i<n_{\text{exp},j}}\}_{0\le j<n}$ by (17) and (18) and (19)

(25) $\mathsf{fv}(\mathtt{match}[n](e'';\{r'_i\}_{1\le i\le n})) \subset \text{dom}(\Delta) \cup \text{dom}(\Gamma) \cup \{t_i\}_{0\le i<n'_{\text{ty}}} \cup_{0\le j<n} \{t_{i,j}\}_{0\le i<n_{\text{ty},j}} \cup \{x_i\}_{0\le i<n'_{\text{exp}}} \cup_{0\le j<n} \{x_{i,j}\}_{0\le i<n_{\text{exp},j}}$ by (23) and (24)

The conclusions hold as follows:

(a) (8) $\cup \bigcup_{0\le j<n}$ (13)$_j$

(b) (9) $\cup \bigcup_{0\le j<n}$ (14)$_j$

(c) (10) $\cup \bigcup_{0\le j<n}$ (15)$_j$

(d) Choose:

i. $\mathtt{match}[n](e'';\{r'_i\}_{1\le i\le n})$

ii. $\{t_i\}_{0\le i<n'_{\text{ty}}} \cup \{\{t_{i,j}\}_{0\le i<n_{\text{ty},j}}\}_{0\le j<n}$; and

iii. $\{x_i\}_{0\le i<n'_{\text{exp}}} \cup \{\{x_{i,j}\}_{0\le i<n_{\text{exp},j}}\}_{0\le j<n}$; and

We have $e = [\{\tau'_i/t_i\}_{0\le i<n'_{\text{ty}}} \cup \{\{\tau_{i,j}/t_{i,j}\}_{0\le i<n_{\text{ty},j}}\}_{0\le j<n}, \{e_i/x_i\}_{0\le i<n'_{\text{exp}}} \cup \{\{e_{i,j}/x_{i,j}\}_{0\le i<n_{\text{exp},j}}\}_{0\le j<n}]\mathtt{match}[n](e'';\{r'_i\}_{1\le i\le n})$ by (22).

(e) (25)

2. By rule induction over the rule typing assumption. There is only one case. In the following, let $\hat{\Delta} = \langle \mathcal{D}; \Delta_{\text{app}}\rangle$ and $\hat{\Gamma} = \langle \mathcal{G}; \Gamma_{\text{app}}\rangle$ and $\mathbb{E} = \hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; \hat{\Phi}; b$.

Case (B.11).

(1) $\grave{r} = \mathtt{prrule}(p.\grave{e})$ by assumption

(2) $r = \mathtt{rule}(p.e)$ by assumption

(3) $\Delta \vdash p : \tau \dashv\!\!| \Gamma'$ by assumption

(4) $\Delta\ \Gamma \cup \Gamma' \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau'$ by assumption

(5) $\mathsf{summary}(\grave{r}) = \mathsf{summary}(\grave{e})$ by definition

(6) $\mathsf{summary}(\grave{e}) = \{\mathtt{splicedt}[m'_i;n'_i]\}_{0\le i<n_{\text{ty}}} \cup \{\mathtt{splicede}[m_i;n_i;\grave{\tau}_i]\}_{0\le i<n_{\text{exp}}}$ by definition

(7) $\{\langle \mathcal{D}; \Delta_{\text{app}}\rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b;m'_i;n'_i)) \leadsto \tau'_i\ \mathsf{type}\}_{0\le i<n_{\text{ty}}}$ by IH, part 1 on (4) and (6)

(8) $\{\emptyset \vdash^{\langle \mathcal{D};\Delta_{\text{app}}\rangle;b} \grave{\tau}_i \leadsto \tau_i\ \mathsf{type}\}_{0\le i<n_{\text{exp}}}$ by IH, part 1 on (4) and (6)

(9) $\{\langle \mathcal{D}; \Delta_{\text{app}}\rangle\ \langle \mathcal{G}; \Gamma_{\text{app}}\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \mathsf{parseUExp}(\mathsf{subseq}(b;m_i;n_i)) \leadsto e_i : \tau_i\}_{0\le i<n_{\text{exp}}}$ by IH, part 1 on (4) and (6)

(10) $e = [\{\tau'_i/t_i\}_{0\le i<n_{\text{ty}}}, \{e_i/x_i\}_{0\le i<n_{\text{exp}}}]e'$ for some e' and fresh $\{t_i\}_{0\le i<n_{\text{ty}}}$ and fresh $\{x_i\}_{0\le i<n_{\text{exp}}}$ by IH, part 1 on (4) and (6)

(11) $\mathsf{fv}(e') \subset \text{dom}(\Delta) \cup \text{dom}(\Gamma) \cup \text{dom}(\Gamma') \cup \{t_i\}_{0\le i<n_{\text{ty}}} \cup \{x_i\}_{0\le i<n_{\text{exp}}}$

(12) $r = [\{\tau_i'/t_i\}_{0\leq i<n_{\text{ty}}}, \{e_i/x_i\}_{0\leq i<n_{\text{exp}}}]\texttt{rule}(p.e')$ by IH, part 1 on (4) and (6) by substitution properties and (10)

(13) $\text{fv}(p) = \text{dom}(\Gamma')$ by Lemma B.5 on (3)

(14) $\text{fv}(\texttt{rule}(p.e')) \subset \text{dom}(\Delta) \cup \text{dom}(\Gamma) \cup \{t_i\}_{0\leq i<n_{\text{ty}}} \cup \{x_i\}_{0\leq i<n_{\text{exp}}}$ by definition of $\text{fv}(r)$ and (11) and (13)

The conclusions hold as follows:

(a) (7)

(b) (8)

(c) (9)

(d) Choosing $\texttt{rule}(p.e')$ and $\{t_i\}_{0\leq i<n_{\text{ty}}}$ and $\{x_i\}_{0\leq i<n_{\text{exp}}}$, by (12)

(e) (14)

□

Theorem B.31 (seTSM Abstract Reasoning Principles). *If* $\langle\mathcal{D};\Delta\rangle\ \langle\mathcal{G};\Gamma\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{a}\ \text{`}b\text{`} \leadsto e : \tau$ *then:*

1. ***(Typing 1)*** $\hat{\Psi} = \hat{\Psi}', \hat{a} \leadsto a \hookrightarrow \texttt{setsm}(\tau; e_{parse})$ *and* $\Delta\ \Gamma \vdash e : \tau$
2. $b \downarrow_{\text{Body}} e_{body}$
3. $e_{parse}(e_{body}) \Downarrow \texttt{inj}[\texttt{SuccessE}](e_{proto})$
4. $e_{proto} \uparrow_{\text{PrExpr}} \grave{e}$
5. ***(Segmentation)*** $\text{seg}(\grave{e})$ segments b
6. $\text{summary}(\grave{e}) = \{\texttt{splicedt}[m_i'; n_i']\}_{0\leq i<n_{ty}} \cup \{\texttt{splicede}[m_i; n_i; \grave{\tau}_i]\}_{0\leq i<n_{exp}}$
7. ***(Typing 2)*** $\{\langle\mathcal{D};\Delta\rangle \vdash \text{parseUTyp}(\text{subseq}(b; m_i'; n_i')) \leadsto \tau_i'\ \text{type}\}_{0\leq i<n_{ty}}$ *and* $\{\Delta \vdash \tau_i'\ \text{type}\}_{0\leq i<n_{ty}}$
8. ***(Typing 3)*** $\{\emptyset \vdash^{\langle\mathcal{D};\Delta\rangle; b} \grave{\tau}_i \leadsto \tau_i\ \text{type}\}_{0\leq i<n_{exp}}$ *and* $\{\Delta \vdash \tau_i\ \text{type}\}_{0\leq i<n_{exp}}$
9. ***(Typing 4)*** $\{\langle\mathcal{D};\Delta\rangle\ \langle\mathcal{G};\Gamma\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \text{parseUExp}(\text{subseq}(b; m_i; n_i)) \leadsto e_i : \tau_i\}_{0\leq i<n_{exp}}$ *and* $\{\Delta\ \Gamma \vdash e_i : \tau_i\}_{0\leq i<n_{exp}}$
10. ***(Capture Avoidance)*** $e = [\{\tau_i'/t_i\}_{0\leq i<n_{ty}}, \{e_i/x_i\}_{0\leq i<n_{exp}}]e'$ *for some* $\{t_i\}_{0\leq i<n_{ty}}$ *and* $\{x_i\}_{0\leq i<n_{exp}}$ *and* e'
11. ***(Context Independence)*** $\text{fv}(e') \subset \{t_i\}_{0\leq i<n_{ty}} \cup \{x_i\}_{0\leq i<n_{exp}}$

Proof. By rule induction over Rules (B.6). There is only one rule that applies. In the following, let $\hat{\Delta} = \langle\mathcal{D};\Delta\rangle$ and $\hat{\Gamma} = \langle\mathcal{G};\Gamma\rangle$.

Case (B.60).

(1) $\hat{\Psi} = \hat{\Psi}', \hat{a} \leadsto a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}})$ by assumption

(2) $\langle\mathcal{D};\Delta\rangle\ \langle\mathcal{G};\Gamma\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{a}\ \text{`}b\text{`} \leadsto e : \tau$ by assumption

(3) $\Delta\ \Gamma \vdash e : \tau$ by Theorem B.28 on (2)

(4) $b \downarrow_{\text{Body}} e_{\text{body}}$ by assumption

(5) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}[\texttt{SuccessE}](e_{\text{proto}})$ by assumption

(6) $e_{\text{proto}} \uparrow_{\text{PrExpr}} \grave{e}$ by assumption

(7) $\mathsf{seg}(\grave{e})$ segments b by assumption

(8) $\emptyset\,\emptyset \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e : \tau$ by assumption

(9) $\mathsf{summary}(\grave{e}) = \{\mathtt{splicedt}[m'_i; n'_i]\}_{0 \le i < n_{\text{ty}}} \cup \{\mathtt{splicede}[m_i; n_i; \grave{\tau}_i]\}_{0 \le i < n_{\text{exp}}}$ by definition

(10) $\{\langle\mathcal{D};\Delta\rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow \tau'_i\ \mathsf{type}\}_{0 \le i < n_{\text{ty}}}$ by Lemma B.30 on (8) and (9)

(11) $\{\Delta \vdash \tau'_i\ \mathsf{type}\}_{0 \le i < n_{\text{ty}}}$ by Lemma B.24, part 1 over (10)

(12) $\{\emptyset \vdash^{\langle\mathcal{D};\Delta\rangle;b} \grave{\tau}_i \rightsquigarrow \tau_i\ \mathsf{type}\}_{0 \le i < n_{\text{exp}}}$ by Lemma B.30 on (8) and (9)

(13) $\emptyset \cap \Delta = \emptyset$ by definition

(14) $\{\Delta \vdash \tau_i\ \mathsf{type}\}_{0 \le i < n_{\text{exp}}}$ by Lemma B.24, part 2 over (12) and (13)

(15) $\{\langle\mathcal{D};\Delta\rangle\ \langle\mathcal{G};\Gamma\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \mathsf{parseUExp}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow e_i : \tau_i\}_{0 \le i < n_{\text{exp}}}$ by Lemma B.30 on (8) and (9)

(16) $\{\Delta\,\Gamma \vdash e_i : \tau_i\}_{0 \le i < n_{\text{exp}}}$ by Theorem B.28 over (15)

(17) $e = [\{\tau'_i/t_i\}_{0 \le i < n_{\text{ty}}}, \{e_i/x_i\}_{0 \le i < n_{\text{exp}}}]e'$ for some e' and fresh $\{t_i\}_{0 \le i < n_{\text{ty}}}$ and fresh $\{x_i\}_{0 \le i < n_{\text{exp}}}$ by Lemma B.30 on (8) and (9)

(18) $\mathsf{fv}(e') \subset \{t_i\}_{0 \le i < n_{\text{ty}}} \cup \{x_i\}_{0 \le i < n_{\text{exp}}}$ by Lemma B.30 on (8) and (9)

The conclusions hold as follows:

1. (1) and (3)
2. (4)
3. (5)
4. (6)
5. (7)
6. (9)
7. (10) and (11)
8. (12) and (14)
9. (15) and (16)
10. (17)
11. (18)

□

Lemma B.32 (Proto-Pattern Expansion Decomposition). *If* $\grave{p} \rightsquigarrow p : \tau \dashv\!|^{\hat{\Delta};\hat{\Phi};b}\ \hat{\Gamma}$ *where*

$$\mathsf{summary}(\grave{p}) = \{\mathtt{splicedt}[m'_i; n'_i]\}_{0 \le i < n_{ty}} \cup \{\mathtt{splicedp}[m_i; n_i; \grave{\tau}_i]\}_{0 \le i < n_{pat}}$$

then all of the following hold:

1. $\{\hat{\Delta} \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow \tau'_i \ \mathsf{type}\}_{0 \le i < n_{ty}}$
2. $\{\emptyset \vdash^{\hat{\Delta}; b} \grave{\tau}_i \rightsquigarrow \tau_i \ \mathsf{type}\}_{0 \le i < n_{pat}}$
3. $\{\hat{\Delta} \vdash_{\hat{\Phi}} \mathsf{parseUPat}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow p_i : \tau_i \dashv\!\vert\, \hat{\Gamma}_i\}_{0 \le i < n_{pat}}$
4. $\hat{\Gamma} = \biguplus_{0 \le i < n_{pat}} \hat{\Gamma}_i$

Proof. By rule induction over Rules (B.12). In the following, let $\mathbb{P} = \hat{\Delta}; \hat{\Phi}; b$.

Case (B.12a).

(1) $\grave{p} = \mathtt{prwildp}$ by assumption
(2) $e = \mathtt{wildp}$ by assumption
(3) $\hat{\Gamma} = \langle \emptyset; \emptyset \rangle$ by assumption
(4) $\mathsf{summary}(\mathtt{prwildp}) = \emptyset$ by definition

The conclusions hold as follows:

1. This conclusion holds trivially because $n_{\text{ty}} = 0$.
2. This conclusion holds trivially because $n_{\text{pat}} = 0$.
3. This conclusion holds trivially because $n_{\text{pat}} = 0$.
4. This conclusion holds trivially because $\hat{\Gamma} = \emptyset$ and $n_{\text{pat}} = 0$.

Case (B.12b).

(1) $\grave{p} = \mathtt{prfoldp}(\grave{p}')$ by assumption
(2) $p = \mathtt{foldp}(p')$ by assumption
(3) $\tau = \mathtt{rec}(t.\tau')$ by assumption
(4) $\grave{p} \rightsquigarrow p : [\mathtt{rec}(t.\tau')/t]\tau' \dashv\!\vert^{\mathbb{P}}\, \hat{\Gamma}$ by assumption
(5) $\mathsf{summary}(\mathtt{prfoldp}(\grave{p}')) = \mathsf{summary}(\grave{p}')$ by definition
(6) $\mathsf{summary}(\grave{p}') = \{\mathtt{splicedt}[m'_i; n'_i]\}_{0 \le i < n_{\text{ty}}} \cup \{\mathtt{splicedp}[m_i; n_i; \grave{\tau}_i]\}_{0 \le i < n_{\text{pat}}}$ by definition
(7) $\{\hat{\Delta} \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow \tau'_i \ \mathsf{type}\}_{0 \le i < n_{\text{ty}}}$ by IH on (4) and (6)
(8) $\{\emptyset \vdash^{\hat{\Delta}; b} \grave{\tau}_i \rightsquigarrow \tau_i \ \mathsf{type}\}_{0 \le i < n_{\text{pat}}}$ by IH on (4) and (6)
(9) $\{\hat{\Delta} \vdash_{\hat{\Phi}} \mathsf{parseUPat}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow p_i : \tau_i \dashv\!\vert\, \hat{\Gamma}_i\}_{0 \le i < n_{\text{pat}}}$ by IH on (4) and (6)
(10) $\hat{\Gamma} = \biguplus_{0 \le i < n_{\text{pat}}} \hat{\Gamma}_i$ by IH on (4) and (6)

The conclusions hold as follows:

1. (7)
2. (8)
3. (9)
4. (10)

Case (B.12c).

(1) $\grave{p} = \mathtt{prtplp}[L](\{j \hookrightarrow \grave{p}_j\}_{j \in L})$ by assumption
(2) $p = \mathtt{tplp}[L](\{j \hookrightarrow p_j\}_{j \in L})$ by assumption
(3) $\tau = \mathtt{prod}[L](\{j \hookrightarrow \tau_j\}_{j \in L})$ by assumption
(4) $\hat{\Gamma} = \biguplus_{j \in L} \hat{\Gamma}_j$ by assumption

(5) $\{\grave{p}_j \rightsquigarrow p_j : \tau_j \dashv\!\Vert^{\mathbb{P}} \hat{\Gamma}_j\}_{j\in L}$ by assumption

(6) $\mathsf{summary}(\mathtt{prtplp}[L](\{j \hookrightarrow \grave{p}_j\}_{j\in L})) = \bigcup_{j\in L} \mathsf{summary}(\grave{p}_j)$ by definition

(7) $\{\mathsf{summary}(\grave{p}_j) = \{\mathtt{splicedt}[m'_{i,j}; n'_{i,j}]\}_{0\le i<n_{\text{ty},j}} \cup \{\mathtt{splicedp}[m_{i,j}; n_{i,j}; \grave{\tau}_{i,j}]\}_{0\le i<n_{\text{pat},j}}\}_{j\in L}$ by definition

(8) $n_{\text{pat}} = \Sigma_{j\in L} n_{\text{pat},j}$ by definition

(9) $\{\{\hat{\Delta} \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_{i,j}; n'_{i,j})) \rightsquigarrow \tau'_{i,j}\ \mathsf{type}\}_{0\le i<n_{\text{ty},j}}\}_{j\in L}$ by IH over (5) and (7)

(10) $\{\{\emptyset \vdash^{\hat{\Delta}; b} \grave{\tau}_{i,j} \rightsquigarrow \tau_{i,j}\ \mathsf{type}\}_{0\le i<n_{\text{pat},j}}\}_{j\in L}$ by IH over (5) and (7)

(11) $\{\{\hat{\Delta} \vdash_{\hat{\Phi}} \mathsf{parseUPat}(\mathsf{subseq}(b; m_{i,j}; n_{i,j})) \rightsquigarrow p_{i,j} : \tau_{i,j} \dashv\!\Vert \hat{\Gamma}_{i,j}\}_{0\le i<n_{\text{pat},j}}\}_{j\in L}$ by IH over (5) and (7)

(12) $\{\hat{\Gamma}_j = \biguplus_{0\le i<n_{\text{pat},j}} \hat{\Gamma}_{i,j}\}_{j\in L}$ by IH over (5) and (7)

(13) $\biguplus_{j\in L} \hat{\Gamma}_j = \biguplus_{j\in L} \biguplus_{i\in n_{\text{pat},j}} \hat{\Gamma}_{i,j}$ by definition and (12)

The conclusions hold as follows:

1. $\bigcup_{j\in L} \bigcup_{i\in n_{\text{ty},j}} (9)_{i,j}$
2. $\bigcup_{j\in L} \bigcup_{i\in n_{\text{pat},j}} (10)_{i,j}$
3. $\bigcup_{j\in L} \bigcup_{i\in n_{\text{pat},j}} (11)_{i,j}$
4. (13)

Case (B.12d).

(1) $\grave{p} = \mathtt{prinjp}[\ell](\grave{p}')$ by assumption

(2) $p = \mathtt{injp}[\ell](p')$ by assumption

(3) $\tau = \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i\in L}; \ell \hookrightarrow \tau')$ by assumption

(4) $\grave{p} \rightsquigarrow p : \tau' \dashv\!\Vert^{\mathbb{P}} \hat{\Gamma}$ by assumption

(5) $\mathsf{summary}(\mathtt{prinjp}[\ell](\grave{p}')) = \mathsf{summary}(\grave{p}')$ by definition

(6) $\mathsf{summary}(\grave{p}') = \{\mathtt{splicedt}[m'_i; n'_i]\}_{0\le i<n_{\text{ty}}} \cup \{\mathtt{splicedp}[m_i; n_i; \grave{\tau}_i]\}_{0\le i<n_{\text{pat}}}$ by definition

(7) $\{\hat{\Delta} \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow \tau'_i\ \mathsf{type}\}_{0\le i<n_{\text{ty}}}$ by IH on (4) and (6)

(8) $\{\emptyset \vdash^{\hat{\Delta}; b} \grave{\tau}_i \rightsquigarrow \tau_i\ \mathsf{type}\}_{0\le i<n_{\text{pat}}}$ by IH on (4) and (6)

(9) $\{\hat{\Delta} \vdash_{\hat{\Phi}} \mathsf{parseUPat}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow p_i : \tau_i \dashv\!\Vert \hat{\Gamma}_i\}_{0\le i<n_{\text{pat}}}$ by IH on (4) and (6)

(10) $\hat{\Gamma} = \biguplus_{0\le i<n_{\text{pat}}} \hat{\Gamma}_i$ by IH on (4) and (6)

The conclusions hold as follows:

1. (7)
2. (8)
3. (9)
4. (10)

Case (B.12e).

(1) $\grave{p} = \texttt{splicedp}[m; n; \grave{\tau}]$ by assumption

(2) $\emptyset \vdash^{\hat{\Delta}; b} \grave{\tau} \leadsto \tau$ type by assumption

(3) $\mathsf{parseUPat}(\mathsf{subseq}(b; m; n)) = \hat{p}$ by assumption

(4) $\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \leadsto p : \tau \dashv\!\!| \hat{\Gamma}$ by assumption

(5) $\mathsf{summary}(\texttt{splicedp}[m; n; \grave{\tau}]) = \mathsf{summary}(\grave{\tau}) \cup \{\texttt{splicedp}[m; n; \grave{\tau}]\}$ by definition

(6) $\mathsf{summary}(\grave{\tau}) = \{\texttt{splicedt}[m'_i; n'_i]\}_{0 \leq i < n_{\text{ty}}}$ by definition

(7) $\{\langle \mathcal{D}; \Delta_{\text{app}} \rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m_i; n_i)) \leadsto \tau_i \text{ type}\}_{0 \leq i < n}$ by Lemma B.29 on (2) and (6)

The conclusions hold as follows:

1. (7)
2. (2)
3. (3) and (4)
4. This conclusion holds by (4) because $n_{\text{pat}} = 1$.

□

Theorem B.33 (spTSM Abstract Reasoning Principles). *If $\hat{\Delta} \vdash_{\hat{\Phi}} \hat{a}$ 'b' $\leadsto p : \tau \dashv\!\!| \hat{\Gamma}$ 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**) $\hat{\Phi} = \hat{\Phi}', \hat{a} \leadsto a \hookrightarrow \texttt{sptsm}(\tau; e_{parse})$ and $\Delta \vdash p : \tau \dashv\!\!| \Gamma$*
2. *$b \downarrow_{\mathsf{Body}} e_{body}$*
3. *$e_{parse}(e_{body}) \Downarrow \texttt{inj}[\mathsf{SuccessP}](e_{proto})$*
4. *$e_{proto} \uparrow_{\mathsf{PrPat}} \grave{p}$*
5. *(**Segmentation**) $\mathsf{seg}(\grave{p})$ segments b*
6. *$\mathsf{summary}(\grave{p}) = \{\texttt{splicedt}[n'_i; m'_i]\}_{0 \leq i < n_{ty}} \cup \{\texttt{splicedp}[m_i; n_i; \grave{\tau}_i]\}_{0 \leq i < n_{pat}}$*
7. *(**Typing 2**) $\{\hat{\Delta} \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_i; n'_i)) \leadsto \tau'_i \text{ type}\}_{0 \leq i < n_{ty}}$ and $\{\Delta \vdash \tau'_i \text{ type}\}_{0 \leq i < n_{ty}}$*
8. *(**Typing 3**) $\{\emptyset \vdash^{\hat{\Delta}; b} \grave{\tau}_i \leadsto \tau_i \text{ type}\}_{0 \leq i < n_{pat}}$ and $\{\Delta \vdash \tau_i \text{ type}\}_{0 \leq i < n_{pat}}$*
9. *(**Typing 4**) $\{\hat{\Delta} \vdash_{\hat{\Phi}} \mathsf{parseUPat}(\mathsf{subseq}(b; m_i; n_i)) \leadsto p_i : \tau_i \dashv\!\!| \langle \mathcal{G}_i; \Gamma_i \rangle\}_{0 \leq i < n_{pat}}$ and $\{\Delta \vdash p_i : \tau_i \dashv\!\!| \Gamma_i\}_{0 \leq i < n_{pat}}$*
10. *(**No Hidden Bindings**) $\mathcal{G} = \biguplus_{0 \leq i < n_{pat}} \mathcal{G}_i$ and $\Gamma = \bigcup_{0 \leq i < n_{pat}} \Gamma_i$*

Proof. By rule induction over Rules (B.8). There is only one rule that applies.

Case (B.8f).

(1) $\hat{\Delta} \vdash_{\hat{\Phi}} \hat{a}$ 'b' $\leadsto p : \tau \dashv\!\!| \hat{\Gamma}$ by assumption

(2) $\hat{\Phi} = \hat{\Phi}', \hat{a} \leadsto a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}})$ by assumption

(3) $\Delta \vdash p : \tau \dashv\!\!| \Gamma$ by Theorem B.26 on (1)

(4) $b \downarrow_{\mathsf{Body}} e_{\text{body}}$ by assumption

(5) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}[\mathsf{SuccessP}](e_{\text{proto}})$ by assumption

(6) $e_{\text{proto}} \uparrow_{\mathsf{PrPat}} \grave{p}$ by assumption

(7) $\mathsf{seg}(\grave{p})$ segments b by assumption

(8) $\grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\hat{\Delta};\hat{\Phi};b} \hat{\Gamma}$ by assumption

(9) $\mathsf{summary}(\grave{p}) = \{\texttt{splicedt}[m'_i; n'_i]\}_{0 \leq i < n_{\mathrm{ty}}} \cup \{\texttt{splicedp}[m_i; n_i;]\}_{0 \leq i < n_{\mathrm{pat}}}$ by definition

(10) $\{\hat{\Delta} \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow \grave{\tau}'_i\ \mathsf{type}\}_{0 \leq i < n_{\mathrm{ty}}}$ by Lemma B.32 on (8) and (9)

(11) $\{\Delta \vdash \tau'_i\ \mathsf{type}\}_{0 \leq i < n_{\mathrm{ty}}}$ by Lemma B.24, part 1 over (10)

(12) $\{\emptyset \vdash^{\hat{\Delta};b} \grave{\tau}_i \rightsquigarrow \tau_i\ \mathsf{type}\}_{0 \leq i < n_{\mathrm{pat}}}$ by Lemma B.32 on (8) and (9)

(13) $\{\Delta \vdash \tau_i\ \mathsf{type}\}_{0 \leq i < n_{\mathrm{pat}}}$ by Lemma B.24, part 2 over (12)

(14) $\{\hat{\Delta} \vdash_{\hat{\Phi}} \mathsf{parseUPat}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow p_i : \tau_i \dashv\!\mid \hat{\Gamma}_i\}_{0 \leq i < n_{\mathrm{pat}}}$ by Lemma B.32 on (8) and (9)

(15) $\{\Delta \vdash p_i : \tau_i \dashv\!\mid \Gamma_i\}_{0 \leq i < n_{\mathrm{pat}}}$ by Theorem B.26 over (14)

(16) $\mathcal{G} = \biguplus_{0 \leq i < n_{\mathrm{pat}}} \mathcal{G}_i$ and $\Gamma = \bigcup_{0 \leq i < n_{\mathrm{pat}}} \Gamma_i$ by Lemma B.32 on (8) and (9)

The conclusions hold as follows:

1. (2) and (3)
2. (4)
3. (5)
4. (6)
5. (7)
6. (9)
7. (10) and (11)
8. (12) and (13)
9. (14) and (15)
10. (16)

□

Appendix C

miniVerse$_{\mathbf{P}}$

C.1 Expanded Language (XL)

C.1.1 Syntax

Signatures and Module Expressions

Sort			**Operational Form**	**Description**
Sig	σ	$::=$	$\mathtt{sig}\{\kappa\}(u.\tau)$	signature
Mod	M	$::=$	X	module variable
			$\mathtt{struct}(c;e)$	structure
			$\mathtt{seal}\{\sigma\}(M)$	seal
			$\mathtt{mlet}\{\sigma\}(M;X.M)$	definition

Kinds and Constructions

Sort			**Operational Form**	**Description**
Kind	κ	$::=$	k	kind variable
			$\mathtt{darr}(\kappa;u.\kappa)$	dependent function
			$\mathtt{unit}$	nullary product
			$\mathtt{dprod}(\kappa;u.\kappa)$	dependent product
			$\mathtt{Type}$	type
			$\mathtt{S}(\tau)$	singleton
Con	c, τ	$::=$	u	construction variable
			t	type variable
			$\mathtt{abs}(u.c)$	abstraction
			$\mathtt{app}(c;c)$	application
			$\mathtt{triv}$	trivial
			$\mathtt{pair}(c;c)$	pair
			$\mathtt{prl}(c)$	left projection
			$\mathtt{prr}(c)$	right projection
			$\mathtt{parr}(\tau;\tau)$	partial function
			$\mathtt{all}\{\kappa\}(u.\tau)$	polymorphic
			$\mathtt{rec}(t.\tau)$	recursive
			$\mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})$	labeled product
			$\mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i\in L})$	labeled sum
			$\mathtt{con}(M)$	construction component

Expressions, Rules and Patterns

Sort			Operational Form	Description
Exp	e	$::=$	x	variable
			$\texttt{lam}\{\tau\}(x.e)$	abstraction
			$\texttt{ap}(e;e)$	application
			$\texttt{clam}\{\kappa\}(u.e)$	construction abstraction
			$\texttt{cap}\{\kappa\}(e)$	construction application
			$\texttt{fold}(e)$	fold
			$\texttt{unfold}(e)$	unfold
			$\texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L})$	labeled tuple
			$\texttt{prj}[\ell](e)$	projection
			$\texttt{inj}[\ell](e)$	injection
			$\texttt{match}[n](e;\{r_i\}_{1\le i\le n})$	match
			$\texttt{val}(M)$	value component
Rule	r	$::=$	$\texttt{rule}(p.e)$	rule
Pat	p	$::=$	x	variable pattern
			$\texttt{wildp}$	wildcard pattern
			$\texttt{foldp}(p)$	fold pattern
			$\texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i\in L})$	labeled tuple pattern
			$\texttt{injp}[\ell](p)$	injection pattern

C.1.2 Statics

Unified Contexts

A *unified context*, Ω, is an ordered finite function. We write

- $\Omega, X : \sigma$ when $X \notin \text{dom}(\Omega)$ for the extension of Ω with a mapping from X to the hypothesis $X : \sigma$.
- $\Omega, x : \tau$ when $x \notin \text{dom}(\Omega)$ for the extension of Ω with a mapping from x to the hypothesis $x : \tau$.
- $\Omega, u :: \kappa$ when $u \notin \text{dom}(\Omega)$ for the extension of Ω with a mapping from u to the hypothesis $u :: \kappa$.

Signatures and Structures

$\boxed{\Omega \vdash \sigma \ \mathsf{sig}}$ σ is a signature

$$\frac{\Omega \vdash \kappa \ \mathsf{kind} \qquad \Omega, u :: \kappa \vdash \tau :: \mathsf{Type}}{\Omega \vdash \texttt{sig}\{\kappa\}(u.\tau)\ \mathsf{sig}} \tag{C.1}$$

$\boxed{\Omega \vdash \sigma \equiv \sigma'}$ σ and σ' are definitionally equal

$$\frac{\Omega \vdash \kappa \equiv \kappa' \qquad \Omega, u :: \kappa \vdash \tau \equiv \tau' :: \mathsf{Type}}{\Omega \vdash \texttt{sig}\{\kappa\}(u.\tau) \equiv \texttt{sig}\{\kappa'\}(u.\tau')} \tag{C.2}$$

$\boxed{\Omega \vdash \sigma <: \sigma'}$ σ is a subsignature of σ'

$$\frac{\Omega \vdash \kappa <:: \kappa' \qquad \Omega, u :: \kappa \vdash \tau <: \tau'}{\Omega \vdash \mathtt{sig}\{\kappa\}(u.\tau) <: \mathtt{sig}\{\kappa'\}(u.\tau')} \tag{C.3}$$

$\boxed{\Omega \vdash M : \sigma}$ M matches σ

$$\frac{\Omega \vdash M : \sigma \qquad \Omega \vdash \sigma <: \sigma'}{\Omega \vdash M : \sigma'} \tag{C.4a}$$

$$\frac{}{\Omega, X : \sigma \vdash X : \sigma} \tag{C.4b}$$

$$\frac{\Omega \vdash c :: \kappa \qquad \Omega \vdash e : [c/u]\tau}{\Omega \vdash \mathtt{struct}(c;e) : \mathtt{sig}\{\kappa\}(u.\tau)} \tag{C.4c}$$

$$\frac{\Omega \vdash \sigma \ \mathsf{sig} \qquad \Omega \vdash M : \sigma}{\Omega \vdash \mathtt{seal}\{\sigma\}(M) : \sigma} \tag{C.4d}$$

$$\frac{\Omega \vdash M : \sigma \qquad \Omega \vdash \sigma' \ \mathsf{sig} \qquad \Omega, X : \sigma \vdash M' : \sigma'}{\Omega \vdash \mathtt{mlet}\{\sigma'\}(M; X.M') : \sigma'} \tag{C.4e}$$

$\boxed{\Omega \vdash M \ \mathsf{mval}}$ M is, or stands for, a module value

$$\frac{}{\Omega \vdash \mathtt{struct}(c;e) \ \mathsf{mval}} \tag{C.5a}$$

$$\frac{}{\Omega, X : \sigma \vdash X \ \mathsf{mval}} \tag{C.5b}$$

Kinds and Constructions

$\boxed{\Omega \vdash \kappa \ \mathsf{kind}}$ κ is a kind

$$\frac{\Omega \vdash \kappa_1 \ \mathsf{kind} \qquad \Omega, u :: \kappa_1 \vdash \kappa_2 \ \mathsf{kind}}{\Omega \vdash \mathtt{darr}(\kappa_1; u.\kappa_2) \ \mathsf{kind}} \tag{C.6a}$$

$$\frac{}{\Omega \vdash \mathtt{unit} \ \mathsf{kind}} \tag{C.6b}$$

$$\frac{\Omega \vdash \kappa_1 \ \mathsf{kind} \qquad \Omega, u :: \kappa_1 \vdash \kappa_2 \ \mathsf{kind}}{\Omega \vdash \mathtt{dprod}(\kappa_1; u.\kappa_2) \ \mathsf{kind}} \tag{C.6c}$$

$$\frac{}{\Omega \vdash \mathtt{Type} \ \mathsf{kind}} \tag{C.6d}$$

$$\frac{\Omega \vdash \tau :: \mathsf{Type}}{\Omega \vdash \mathtt{S}(\tau)\ \mathsf{kind}} \tag{C.6e}$$

$\boxed{\Omega \vdash \kappa \equiv \kappa'}$ κ and κ' are definitionally equal

$$\frac{\Omega \vdash \kappa\ \mathsf{kind}}{\Omega \vdash \kappa \equiv \kappa} \tag{C.7a}$$

$$\frac{\Omega \vdash \kappa \equiv \kappa'}{\Omega \vdash \kappa' \equiv \kappa} \tag{C.7b}$$

$$\frac{\Omega \vdash \kappa \equiv \kappa' \qquad \Omega \vdash \kappa' \equiv \kappa''}{\Omega \vdash \kappa \equiv \kappa''} \tag{C.7c}$$

$$\frac{\Omega \vdash \kappa_1 \equiv \kappa_1' \qquad \Omega, u :: \kappa_1 \vdash \kappa_2 \equiv \kappa_2'}{\Omega \vdash \mathtt{darr}(\kappa_1; u.\kappa_2) \equiv \mathtt{darr}(\kappa_1'; u.\kappa_2')} \tag{C.7d}$$

$$\frac{\Omega \vdash \kappa_1 \equiv \kappa_1' \qquad \Omega, u :: \kappa_1 \vdash \kappa_2 \equiv \kappa_2'}{\Omega \vdash \mathtt{dprod}(\kappa_1; u.\kappa_2) \equiv \mathtt{dprod}(\kappa_1'; u.\kappa_2')} \tag{C.7e}$$

$$\frac{\Omega \vdash c \equiv c' :: \mathsf{Type}}{\Omega \vdash \mathtt{S}(c) \equiv \mathtt{S}(c')} \tag{C.7f}$$

$\boxed{\Omega \vdash \kappa <:: \kappa'}$ κ is a subkind of κ'

$$\frac{\Omega \vdash \kappa \equiv \kappa'}{\Omega \vdash \kappa <:: \kappa'} \tag{C.8a}$$

$$\frac{\Omega \vdash \kappa <:: \kappa' \qquad \Omega \vdash \kappa' <:: \kappa''}{\Omega \vdash \kappa <:: \kappa''} \tag{C.8b}$$

$$\frac{\Omega \vdash \kappa_1' <:: \kappa_1 \qquad \Omega, u :: \kappa_1' \vdash \kappa_2 <:: \kappa_2'}{\Omega \vdash \mathtt{darr}(\kappa_1; u.\kappa_2) <:: \mathtt{darr}(\kappa_1'; u.\kappa_2')} \tag{C.8c}$$

$$\frac{\Omega \vdash \kappa_1 <:: \kappa_1' \qquad \Omega, u :: \kappa_1 \vdash \kappa_2 <:: \kappa_2'}{\Omega \vdash \mathtt{dprod}(\kappa_1; u.\kappa_2) <:: \mathtt{dprod}(\kappa_1'; u.\kappa_2')} \tag{C.8d}$$

$$\frac{\Omega \vdash \tau :: \mathsf{Type}}{\Omega \vdash \mathtt{S}(\tau) <:: \mathsf{Type}} \tag{C.8e}$$

$$\frac{\Omega \vdash \tau <: \tau'}{\Omega \vdash \mathtt{S}(\tau) <:: \mathtt{S}(\tau')} \tag{C.8f}$$

$\boxed{\Omega \vdash c :: \kappa}$ c has kind κ

$$\frac{\Omega \vdash c :: \kappa_1 \qquad \Omega \vdash \kappa_1 <:: \kappa_2}{\Omega \vdash c :: \kappa_2} \tag{C.9a}$$

$$\frac{}{\Omega, u :: \kappa \vdash u :: \kappa} \tag{C.9b}$$

$$\frac{\Omega, u :: \kappa_1 \vdash c_2 :: \kappa_2}{\Omega \vdash \texttt{abs}(u.c_2) :: \texttt{darr}(\kappa_1; u.\kappa_2)} \tag{C.9c}$$

$$\frac{\Omega \vdash c_1 :: \texttt{darr}(\kappa_2; u.\kappa) \qquad \Omega \vdash c_2 :: \kappa_2}{\Omega \vdash \texttt{app}(c_1; c_2) :: [c_1/u]\kappa} \tag{C.9d}$$

$$\frac{}{\Omega \vdash \texttt{triv} :: \texttt{unit}} \tag{C.9e}$$

$$\frac{\Omega \vdash c_1 :: \kappa_1 \qquad \Omega \vdash c_2 :: [c_1/u]\kappa_2}{\Omega \vdash \texttt{pair}(c_1; c_2) :: \texttt{dprod}(\kappa_1; u.\kappa_2)} \tag{C.9f}$$

$$\frac{\Omega \vdash c :: \texttt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash \texttt{prl}(c) :: \kappa_1} \tag{C.9g}$$

$$\frac{\Omega \vdash c :: \texttt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash \texttt{prr}(c) :: [\texttt{prl}(c)/u]\kappa_2} \tag{C.9h}$$

$$\frac{\Omega \vdash \tau_1 :: \mathsf{Type} \qquad \Omega \vdash \tau_2 :: \mathsf{Type}}{\Omega \vdash \texttt{parr}(\tau_1; \tau_2) :: \mathsf{Type}} \tag{C.9i}$$

$$\frac{\Omega \vdash \kappa \;\mathsf{kind} \qquad \Omega, u :: \kappa \vdash \tau :: \mathsf{Type}}{\Omega \vdash \texttt{all}\{\kappa\}(u.\tau) :: \mathsf{Type}} \tag{C.9j}$$

$$\frac{\Omega, t :: \mathsf{Type} \vdash \tau :: \mathsf{Type}}{\Omega \vdash \texttt{rec}(t.\tau) :: \mathsf{Type}} \tag{C.9k}$$

$$\frac{\{\Omega \vdash \tau_i :: \mathsf{Type}\}_{1 \leq i \leq n}}{\Omega \vdash \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) :: \mathsf{Type}} \tag{C.9l}$$

$$\frac{\{\Omega \vdash \tau_i :: \mathsf{Type}\}_{1 \leq i \leq n}}{\Omega \vdash \texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) :: \mathsf{Type}} \tag{C.9m}$$

$$\frac{\Omega \vdash c :: \mathsf{Type}}{\Omega \vdash c :: \texttt{S}(c)} \tag{C.9n}$$

$$\frac{\Omega \vdash M \;\mathsf{mval} \qquad \Omega \vdash M : \texttt{sig}\{\kappa\}(u.\tau)}{\Omega \vdash \texttt{con}(M) :: \kappa} \tag{C.9o}$$

$\boxed{\Omega \vdash c \equiv c' :: \kappa}$ c and c' are definitionally equal as constructions of kind κ

$$\frac{\Omega \vdash c :: \kappa}{\Omega \vdash c \equiv c :: \kappa} \tag{C.10a}$$

$$\frac{\Omega \vdash c \equiv c' :: \kappa}{\Omega \vdash c' \equiv c :: \kappa} \tag{C.10b}$$

$$\frac{\Omega \vdash c \equiv c' :: \kappa \qquad \Omega \vdash c' \equiv c'' :: \kappa}{\Omega \vdash c \equiv c'' :: \kappa} \tag{C.10c}$$

$$\frac{\Omega, u :: \kappa_1 \vdash c \equiv c' :: \kappa_2}{\Omega \vdash \texttt{abs}(u.c) \equiv \texttt{abs}(u.c') :: \texttt{darr}(\kappa_1; u.\kappa_2)} \tag{C.10d}$$

$$\frac{\Omega \vdash c_1 \equiv c_1' :: \texttt{darr}(\kappa_2; u.\kappa) \qquad \Omega \vdash c_2 \equiv c_2' :: \kappa_2}{\Omega \vdash \texttt{app}(c_1; c_2) \equiv \texttt{app}(c_1'; c_2') :: \kappa} \tag{C.10e}$$

$$\frac{\Omega \vdash \texttt{abs}(u.c) :: \texttt{darr}(\kappa_2; u.\kappa) \qquad \Omega \vdash c_2 :: \kappa_2}{\Omega \vdash \texttt{app}(\texttt{abs}(u.c); c_2) \equiv [c_2/u]c :: [c_2/u]\kappa} \tag{C.10f}$$

$$\frac{\Omega \vdash c_1 \equiv c_1' :: \kappa_1 \qquad \Omega \vdash c_2 \equiv c_2' :: [c_1/u]\kappa_2}{\Omega \vdash \texttt{pair}(c_1; c_2) \equiv \texttt{pair}(c_1'; c_2') :: \texttt{dprod}(\kappa_1; u.\kappa_2)} \tag{C.10g}$$

$$\frac{\Omega \vdash c \equiv c' :: \texttt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash \texttt{prl}(c) \equiv \texttt{prl}(c') :: \kappa_1} \tag{C.10h}$$

$$\frac{\Omega \vdash c_1 :: \kappa_1 \qquad \Omega \vdash c_2 :: \kappa_2}{\Omega \vdash \texttt{prl}(\texttt{pair}(c_1; c_2)) \equiv c_1 :: \kappa_1} \tag{C.10i}$$

$$\frac{\Omega \vdash c \equiv c' :: \texttt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash \texttt{prr}(c) \equiv \texttt{prr}(c') :: [\texttt{prl}(c)/u]\kappa_2} \tag{C.10j}$$

$$\frac{\Omega \vdash c_1 :: \kappa_1 \qquad \Omega \vdash c_2 :: \kappa_2}{\Omega \vdash \texttt{prr}(\texttt{pair}(c_1; c_2)) \equiv c_2 :: \kappa_2} \tag{C.10k}$$

$$\frac{\Omega \vdash \tau_1 \equiv \tau_1' :: \texttt{Type} \qquad \Omega \vdash \tau_2 \equiv \tau_2' :: \texttt{Type}}{\Omega \vdash \texttt{parr}(\tau_1; \tau_2) \equiv \texttt{parr}(\tau_1'; \tau_2') :: \texttt{Type}} \tag{C.10l}$$

$$\frac{\Omega \vdash \kappa \equiv \kappa' \qquad \Omega, u :: \kappa \vdash \tau \equiv \tau' :: \texttt{Type}}{\Omega \vdash \texttt{all}\{\kappa\}(u.\tau) \equiv \texttt{all}\{\kappa'\}(u.\tau') :: \texttt{Type}} \tag{C.10m}$$

$$\frac{\Omega, t :: \texttt{Type} \vdash \tau \equiv \tau' :: \texttt{Type}}{\Omega \vdash \texttt{rec}(t.\tau) \equiv \texttt{rec}(t.\tau') :: \texttt{Type}} \tag{C.10n}$$

$$\frac{\{\Omega \vdash \tau_i \equiv \tau_i' :: \texttt{Type}\}_{1 \leq i \leq n}}{\Omega \vdash \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \equiv \texttt{prod}[L](\{i \hookrightarrow \tau_i'\}_{i \in L}) :: \texttt{Type}} \tag{C.10o}$$

$$\frac{\{\Omega \vdash \tau_i \equiv \tau_i' :: \mathtt{Type}\}_{1 \leq i \leq n}}{\Omega \vdash \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \equiv \mathtt{sum}[L](\{i \hookrightarrow \tau_i'\}_{i \in L}) :: \mathtt{Type}} \tag{C.10p}$$

$$\frac{\Omega \vdash c :: \mathtt{S}(c')}{\Omega \vdash c \equiv c' :: \mathtt{Type}} \tag{C.10q}$$

$$\frac{\Omega \vdash \mathtt{struct}(c;e) : \mathtt{sig}\{\kappa\}(u.\tau)}{\Omega \vdash \mathtt{con}(\mathtt{struct}(c;e)) \equiv c :: \kappa} \tag{C.10r}$$

Expressions, Rules and Patterns

$\boxed{\Omega \vdash \tau <: \tau'}$ τ is a subtype of τ'

$$\frac{\Omega \vdash \tau_1 \equiv \tau_2 :: \mathtt{Type}}{\Omega \vdash \tau_1 <: \tau_2} \tag{C.11a}$$

$$\frac{\Omega \vdash \tau <: \tau' \qquad \Omega \vdash \tau' <: \tau''}{\Omega \vdash \tau <: \tau''} \tag{C.11b}$$

$$\frac{\Omega \vdash \tau_1' <: \tau_1 \qquad \Omega \vdash \tau_2 <: \tau_2'}{\Omega \vdash \mathtt{parr}(\tau_1;\tau_2) <: \mathtt{parr}(\tau_1';\tau_2')} \tag{C.11c}$$

$$\frac{\Omega \vdash \kappa' <:: \kappa \qquad \Omega, u :: \kappa' \vdash \tau <: \tau'}{\Omega \vdash \mathtt{all}\{\kappa\}(u.\tau) <: \mathtt{all}\{\kappa'\}(u.\tau')} \tag{C.11d}$$

$$\frac{\{\Omega \vdash \tau_i <: \tau_i'\}_{i \in L}}{\Omega \vdash \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) <: \mathtt{prod}[L](\{i \hookrightarrow \tau_i'\}_{i \in L})} \tag{C.11e}$$

$$\frac{\{\Omega \vdash \tau_i <: \tau_i'\}_{i \in L}}{\Omega \vdash \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) <: \mathtt{sum}[L](\{i \hookrightarrow \tau_i'\}_{i \in L})} \tag{C.11f}$$

$\boxed{\Omega \vdash e : \tau}$ e has type τ

$$\frac{\Omega \vdash e : \tau \qquad \Omega \vdash \tau <: \tau'}{\Omega \vdash e : \tau'} \tag{C.12a}$$

$$\frac{}{\Omega, x : \tau \vdash x : \tau} \tag{C.12b}$$

$$\frac{\Omega \vdash \tau :: \mathtt{Type} \qquad \Omega, x : \tau \vdash e : \tau'}{\Omega \vdash \mathtt{lam}\{\tau\}(x.e) : \mathtt{parr}(\tau;\tau')} \tag{C.12c}$$

$$\frac{\Omega \vdash e_1 : \mathtt{parr}(\tau;\tau') \qquad \Omega \vdash e_2 : \tau}{\Omega \vdash \mathtt{ap}(e_1;e_2) : \tau'} \tag{C.12d}$$

$$\frac{\Omega \vdash \kappa\ \mathsf{kind} \qquad \Omega, u :: \kappa \vdash e : \tau}{\Omega \vdash \mathtt{clam}\{\kappa\}(u.e) : \mathtt{all}\{\kappa\}(u.\tau)} \tag{C.12e}$$

$$\frac{\Omega \vdash e : \texttt{all}\{\kappa\}(u.\tau) \qquad \Omega \vdash c :: \kappa}{\Omega \vdash \texttt{cap}\{c\}(e) : [c/u]\tau} \tag{C.12f}$$

$$\frac{\Omega \vdash e : [\texttt{rec}(t.\tau)/t]\tau}{\Omega \vdash \texttt{fold}(e) : \texttt{rec}(t.\tau)} \tag{C.12g}$$

$$\frac{\Omega \vdash e : \texttt{rec}(t.\tau)}{\Omega \vdash \texttt{unfold}(e) : [\texttt{rec}(t.\tau)/t]\tau} \tag{C.12h}$$

$$\frac{\{\Omega \vdash e_i : \tau_i\}_{i \in L}}{\Omega \vdash \texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{C.12i}$$

$$\frac{\Omega \vdash e : \texttt{prod}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)}{\Omega \vdash \texttt{prj}[\ell](e) : \tau} \tag{C.12j}$$

$$\frac{\Omega \vdash e : \tau}{\Omega \vdash \texttt{inj}[\ell](e) : \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)} \tag{C.12k}$$

$$\frac{\Omega \vdash e : \tau \qquad \{\Omega \vdash r_i : \tau \Mapsto \tau'\}_{1 \le i \le n}}{\Omega \vdash \texttt{match}[n](e; \{r_i\}_{1 \le i \le n}) : \tau'} \tag{C.12l}$$

$$\frac{\Omega \vdash M \; \mathsf{mval} \qquad \Omega \vdash M : \texttt{sig}\{\kappa\}(u.\tau)}{\Omega \vdash \texttt{val}(M) : [\texttt{con}(M)/u]\tau} \tag{C.12m}$$

$\boxed{\Omega \vdash r : \tau \Mapsto \tau'}$ r takes values of type τ to values of type τ'

$$\frac{\Omega \vdash p : \tau \dashv\!\!\mid \Omega' \qquad \Omega \cup \Omega' \vdash e : \tau'}{\Omega \vdash \texttt{rule}(p.e) : \tau \Mapsto \tau'} \tag{C.13}$$

$\boxed{\Omega \vdash p : \tau \dashv\!\!\mid \Omega'}$ p matches values of type τ generating hypotheses Ω'

$$\frac{\Omega \vdash p : \tau \dashv\!\!\mid \Omega' \qquad \Omega \vdash \tau <: \tau'}{\Omega \vdash p : \tau' \dashv\!\!\mid \Omega'} \tag{C.14a}$$

$$\frac{}{\Omega \vdash x : \tau \dashv\!\!\mid x : \tau} \tag{C.14b}$$

$$\frac{}{\Omega \vdash \texttt{wildp} : \tau \dashv\!\!\mid \emptyset} \tag{C.14c}$$

$$\frac{\Omega \vdash p : [\texttt{rec}(t.\tau)/t]\tau \dashv\!\!\mid \Omega'}{\Omega \vdash \texttt{foldp}(p) : \texttt{rec}(t.\tau) \dashv\!\!\mid \Omega'} \tag{C.14d}$$

$$\frac{\{\Omega \vdash p_i : \tau_i \dashv\!\!\mid \Omega_i\}_{i \in L}}{\Omega \vdash \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\!\!\mid \cup_{i \in L} \Omega_i} \tag{C.14e}$$

$$\frac{\Omega \vdash p : \tau \dashv\!\!\mid \Omega'}{\Omega \vdash \texttt{injp}[\ell](p) : \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\!\!\mid \Omega'} \tag{C.14f}$$

Metatheory

The rules above are syntax-directed, so we assume an inversion lemma for each rule as needed without stating it separately or proving it explicitly. The following standard lemmas also hold, for all basic judgements J above.

Lemma C.1 (Weakening)**.** *If $\Omega \vdash J$ then $\Omega \cup \Omega' \vdash J$.*

Proof Sketch. By straightforward mutual rule induction. ☐

Definition C.2. *A* substitution*, ω, is a finite function that maps:*

- *each $X \in dom(\omega)$ to a module expression subtitution, M/X;*
- *each $u \in dom(\omega)$ to a construction substitution, c/u; and*
- *each $x \in dom(\omega)$ to an expression substitution, e/x.*

We write $\Omega \vdash \omega : \Omega'$ iff $dom(\omega) = dom(\Omega')$ and:

- *for each $M/X \in \omega$, we have $X : \sigma \in \Omega'$ and $\Omega \vdash M : \sigma$ and $\Omega \vdash M\ \mathsf{mval}$; and*
- *for each $c/u \in \omega$, we have $u :: \kappa \in \Omega'$ and $\Omega \vdash c :: \kappa$; and*
- *for each $e/x \in \omega$, we have $x : \tau \in \Omega'$ and $\Omega \vdash e : \tau$.*

We simultaneously apply a substitution by placing it in prefix position. For example, $[\omega]e$ applies the substitutions ω simultaneously to e.

Lemma C.3 (Substitution)**.** *If $\Omega \cup \Omega' \cup \Omega'' \vdash J$ and $\Omega \vdash \omega : \Omega'$ then $\Omega \cup [\omega]\Omega'' \vdash [\omega]J$.*

Proof Sketch. By straightforward rule induction. ☐

Lemma C.4 (Decomposition)**.** *If $\Omega \cup [\omega]\Omega'' \vdash [\omega]J$ and $\Omega \vdash \omega : \Omega'$ then $\Omega \cup \Omega' \cup \Omega'' \vdash J$.*

Proof Sketch. By straightforward rule induction. ☐

Lemma C.5 (Pattern Binding)**.** *If $\Omega \vdash p : \tau \dashV \Omega'$ then $dom(\Omega') = \mathsf{patvars}(p)$.*

Proof Sketch. By straightforward rule induction over Rules (C.14). ☐

C.1.3 Structural Dynamics

The structural dynamics of modules is defined as a transition system, and is organized around judgements of the following form:

Judgement Form	**Description**
$M \mapsto M'$	M transitions to M'
$M\ \mathsf{val}$	M is a module value
$M\ \mathsf{matchfail}$	M raises match failure

The structural dynamics of expressions is also defined as a transition system, and is organized around judgements of the following form:

Judgement Form	**Description**
$e \mapsto e'$	e transitions to e'
$e\ \mathsf{val}$	e is a value
$e\ \mathsf{matchfail}$	e raises match failure

We also define auxiliary judgements for *iterated transition*, $e \mapsto^* e'$, and *evaluation*, $e \Downarrow e'$ of expressions.

Definition C.6 (Iterated Transition)**.** *Iterated transition, $e \mapsto^* e'$, is the reflexive, transitive closure of the transition judgement, $e \mapsto e'$.*

Definition C.7 (Evaluation)**.** $e \Downarrow e'$ *iff* $e \mapsto^* e'$ *and* e' val.

Similarly, we lift these definitions to the level of module expressions as well.

Definition C.8 (Iterated Module Transition)**.** *Iterated transition, $M \mapsto^* M'$, is the reflexive, transitive closure of the transition judgement, $M \mapsto M'$.*

Definition C.9 (Module Evaluation)**.** $M \Downarrow M'$ *iff* $M \mapsto^* M'$ *and* M' val.

As in $\mathsf{miniVerse_S}$, our subsequent developments do not make mention of particular rules in the dynamics, nor do they make mention of other judgements, not listed above, that are used only for defining the dynamics of the match operator, so we do not produce these details here. Instead, it suffices to state the following conditions.

The Preservation condition ensures that evaluation preserves typing.

Condition C.10 (Preservation)**.**

1. *If* $\vdash M : \sigma$ *and* $M \mapsto M'$ *then* $\vdash M : \sigma$.
2. *If* $\vdash e : \tau$ *and* $e \mapsto e'$ *then* $\vdash e' : \tau$.

The Progress condition ensures that evaluation of a well-typed expanded expression cannot "get stuck". We must consider the possibility of match failure in this condition.

Condition C.11 (Progress)**.**

1. *If* $\vdash M : \sigma$ *then either* M val *or* M $\mathsf{matchfail}$ *or there exists an* M' *such that* $M \mapsto M'$.
2. *If* $\vdash e : \tau$ *then either* e val *or* e $\mathsf{matchfail}$ *or there exists an* e' *such that* $e \mapsto e'$.

C.2 Unexpanded Language (UL)

C.2.1 Syntax

Stylized Syntax – Unexpanded Signatures and Modules

Sort			**Stylized Form**	**Description**
USig	$\hat{\sigma}$	$::=$	$[\![\hat{u} :: \hat{\kappa}; \hat{\tau}]\!]$	signature
UMod	$\hat{M}$	$::=$	$\hat{X}$	module identifier
			$[\![\hat{c}; \hat{e}]\!]$	structure
			$\hat{M} \upharpoonright \hat{\sigma}$	seal
			$(\mathtt{let}\ \hat{X} = \hat{M}\ \mathtt{in}\ \hat{M}) : \hat{\sigma}$	definition
			$\mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for}\ \mathtt{expressions}\ \mathtt{by}\ \mathtt{static}\ e\ \mathtt{in}\ \hat{M}$	peTSM definition
			$\mathtt{let}\ \mathtt{syntax}\ \hat{a} = \hat{\epsilon}\ \mathtt{for}\ \mathtt{expressions}\ \mathtt{in}\ \hat{M}$	peTSM binding
			$\mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for}\ \mathtt{patterns}\ \mathtt{by}\ \mathtt{static}\ e\ \mathtt{in}\ \hat{M}$	ppTSM definition
			$\mathtt{let}\ \mathtt{syntax}\ \hat{a} = \hat{\epsilon}\ \mathtt{for}\ \mathtt{patterns}\ \mathtt{in}\ \hat{M}$	ppTSM binding

Stylized Syntax – Unexpanded Kinds and Constructions

Sort			Stylized Form	Description
UKind	$\hat{\kappa}$	$::=$	$(\hat{u} :: \hat{\kappa}) \to \hat{\kappa}$	dependent function
			$\langle\!\langle\rangle\!\rangle$	nullary product
			$(\hat{u} :: \hat{\kappa}) \times \hat{\kappa}$	dependent product
			$\mathtt{T}$	type
			$[=\hat{\tau}]$	singleton
UCon	$\hat{c}, \hat{\tau}$	$::=$	$\hat{u}$	construction identifier
			$\hat{t}$	
			$\hat{c} :: \hat{\kappa}$	ascription
			$\lambda\hat{u}.\hat{c}$	abstraction
			$c(c)$	application
			$\langle\!\langle\rangle\!\rangle$	trivial
			$\langle\!\langle\hat{c}, \hat{c}\rangle\!\rangle$	pair
			$\hat{c} \cdot \mathtt{l}$	left projection
			$\hat{c} \cdot \mathtt{r}$	right projection
			$\hat{\tau} \rightharpoonup \hat{\tau}$	partial function
			$\forall(\hat{u} :: \hat{\kappa}).\hat{\tau}$	polymorphic
			$\mu\hat{t}.\hat{\tau}$	recursive
			$\langle\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}\rangle$	labeled product
			$[\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}]$	labeled sum
			$\hat{X} \cdot \mathtt{c}$	construction component

Stylized Syntax – Unexpanded Expressions, Rules and Patterns

Sort			Stylized Form	Description
UExp	$\hat{e}$	$::=$	$\hat{x}$	identifier
			$\hat{e} : \hat{\tau}$	ascription
			$\mathtt{let}\,\mathtt{val}\; \hat{x} = \hat{e}\; \mathtt{in}\; \hat{e}$	value binding
			$\lambda\hat{x}{:}\hat{\tau}.\hat{e}$	abstraction
			$\hat{e}(\hat{e})$	application
			$\Lambda\hat{u}{::}\hat{\kappa}.\hat{e}$	construction abstraction
			$\hat{e}[\hat{c}]$	construction application
			$\mathtt{fold}(\hat{e})$	fold
			$\mathtt{unfold}(\hat{e})$	unfold
			$\langle\{i \hookrightarrow \hat{e}_i\}_{i\in L}\rangle$	labeled tuple
			$\hat{e} \cdot \ell$	projection
			$\mathtt{inj}[\ell](\hat{e})$	injection
			$\mathtt{match}\; \hat{e}\; \{\hat{r}_i\}_{1\leq i\leq n}$	match
			$\hat{X} \cdot \mathtt{v}$	value component
			$\hat{\epsilon}$ ‘b‘	peTSM application
URule	$\hat{r}$	$::=$	$\hat{p} \Rightarrow \hat{e}$	match rule
UPat	$\hat{p}$	$::=$	$\hat{x}$	identifier pattern
			_	wildcard pattern
			$\mathtt{fold}(\hat{p})$	fold pattern
			$\langle\{i \hookrightarrow \hat{p}_i\}_{i\in L}\rangle$	labeled tuple pattern
			$\mathtt{inj}[\ell](\hat{p})$	injection pattern
			$\hat{\epsilon}$ ‘b‘	ppTSM application

Stylized Syntax – Unexpanded TSM Types and Expressions

Sort			Stylized Form	Description
UMType	$\hat{\rho}$	$::=$	$\hat{\tau}$	type annotation
			$\forall\hat{X}{:}\hat{\sigma}.\hat{\rho}$	module parameterization
UMExp	$\hat{\epsilon}$	$::=$	$\hat{a}$	TSM identifier reference
			$\Lambda\hat{X}{:}\hat{\sigma}.\hat{\epsilon}$	module abstraction
			$\hat{\epsilon}(\hat{X})$	module application

Stylized Syntax – TSM Types and Expressions

Sort			Operational Form	Description
MType	ρ	$::=$	$\mathtt{type}(\tau)$	type annotation
			$\mathtt{allmods}\{\sigma\}(X.\rho)$	module parameterization
MExp	ϵ	$::=$	$\mathtt{defref}[a]$	TSM definition reference
			$\mathtt{absmod}\{\sigma\}(X.\epsilon)$	module abstraction
			$\mathtt{apmod}\{M\}(\epsilon)$	module application

Body Lengths

We write $\|b\|$ for the length of b. The metafunction $\|\hat{M}\|$ computes the sum of the lengths of expression literal bodies in $\hat{M}$:

$$\begin{array}{ll}
\|\hat{X}\| & = 0 \\
\|[\![\hat{c}; \hat{e}]\!]\| & = \|\hat{e}\| \\
\|\hat{M} \upharpoonright \hat{\sigma}\| & = \|\hat{M}\| \\
\|(\mathtt{let}\ \hat{X} = \hat{M}\ \mathtt{in}\ \hat{M}') : \hat{\sigma}\| & = \|\hat{M}\| + \|\hat{M}'\| \\
\|\mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for}\ \mathtt{expressions}\ \mathtt{by}\ \mathtt{static}\ e\ \mathtt{in}\ \hat{M}\| & = \|\hat{M}\| \\
\|\mathtt{let}\ \mathtt{syntax}\ \hat{a} = \hat{e}\ \mathtt{for}\ \mathtt{expressions}\ \mathtt{in}\ \hat{M}\| & = \|\hat{M}\| \\
\|\mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for}\ \mathtt{patterns}\ \mathtt{by}\ \mathtt{static}\ e\ \mathtt{in}\ \hat{M}\| & = \|\hat{M}\| \\
\|\mathtt{let}\ \mathtt{syntax}\ \hat{a} = \hat{e}\ \mathtt{for}\ \mathtt{patterns}\ \mathtt{in}\ \hat{M}\| & = \|\hat{M}\|
\end{array}$$

and $\|\hat{e}\|$ computes the sum of the lengths of expression literal bodies in $\hat{e}$:

$$\begin{array}{ll}
\|\hat{x}\| & = 0 \\
\|\lambda\hat{x}{:}\hat{\tau}.\hat{e}\| & = \|\hat{e}\| \\
\|\hat{e}_1(\hat{e}_2)\| & = \|\hat{e}_1\| + \|\hat{e}_2\| \\
\|\Lambda\hat{u}{::}\hat{\kappa}.\hat{e}\| & = \|\hat{e}\| \\
\|\hat{e}\mathtt{[}\hat{c}\mathtt{]}\| & = \|\hat{e}\| \\
\|\mathtt{fold}(\hat{e})\| & = \|\hat{e}\| \\
\|\mathtt{unfold}(\hat{e})\| & = \|\hat{e}\| \\
\|\langle\{i \hookrightarrow \hat{e}_i\}_{i \in L}\rangle\| & = \sum_{i \in L} \|\hat{e}_i\| \\
\|\ell \cdot \hat{e}\| & = \|\hat{e}\| \\
\|\mathtt{inj}[\ell](\hat{e})\| & = \|\hat{e}\| \\
\|\mathtt{match}\ \hat{e}\ \{\hat{r}_i\}_{1 \leq i \leq n}\| & = \|\hat{e}\| + \sum_{1 \leq i \leq n} \|r_i\| \\
\|\hat{X} \cdot \mathtt{v}\| & = 0 \\
\|\hat{\epsilon}\ \text{‘}b\text{‘}\| & = \|b\|
\end{array}$$

and $\|\hat{r}\|$ computes the sum of the lengths of expression literal bodies in $\hat{r}$:

$$\|\hat{p} \Rightarrow \hat{e}\| = \|\hat{e}\|$$

Similarly, the metafunction $\|\hat{p}\|$ computes the sum of the lengths of the pattern literal bodies in $\hat{p}$:

$$\begin{aligned}
\|\hat{x}\| &= 0 \\
\|\mathtt{fold}(\hat{p})\| &= \|\hat{p}\| \\
\|\langle\{i \hookrightarrow \hat{p}_i\}_{i \in L}\rangle\| &= \sum_{i \in L} \|\hat{p}_i\| \\
\|\mathtt{inj}[\ell](\hat{p})\| &= \|\hat{p}\| \\
\|\hat{\epsilon}\ \text{‘}b\text{‘}\| &= \|b\|
\end{aligned}$$

Common Unexpanded Forms

Each expanded form, with a few minor exceptions noted below, maps onto an unexpanded form. We refer to these as the *common forms*. In particular:

- Each module variable, X, maps onto a unique module identifier, written $\widehat{X}$.
- Each signature, σ, maps onto an unexpanded signature, $\mathcal{U}(\sigma)$, as follows:

$$\mathcal{U}(\texttt{sig\{}\kappa\texttt{\}(}u.c\texttt{)}) = [\![\widehat{u} :: \mathcal{U}(\kappa); \mathcal{U}(c)]\!]$$

- Each module expression, M, maps onto an unexpanded module expression, $\hat{M}$, as follows:

$$\begin{aligned}
\mathcal{U}(X) &= \widehat{X} \\
\mathcal{U}(\texttt{struct(}\hat{c};\hat{e}\texttt{)}) &= [\![\mathcal{U}(\hat{c});\mathcal{U}(\hat{e})]\!] \\
\mathcal{U}(\texttt{seal\{}\sigma\texttt{\}(}M\texttt{)}) &= \mathcal{U}(M) \upharpoonright \mathcal{U}(\sigma) \\
\mathcal{U}(\texttt{mlet\{}\sigma\texttt{\}(}M; X.M'\texttt{)}) &= (\texttt{let}\ \widehat{X} = \mathcal{U}(M)\ \texttt{in}\ \mathcal{U}(M')) : \mathcal{U}(\sigma)
\end{aligned}$$

- Each construction variable, u, maps onto a unique type identifier, written $\widehat{u}$.
- Each kind, κ, maps onto an unexpanded kind, $\mathcal{U}(\kappa)$, as follows:

$$\begin{aligned}
\mathcal{U}(\texttt{darr(}\kappa; u.\kappa'\texttt{)}) &= (\widehat{u} :: \mathcal{U}(\kappa)) \to \mathcal{U}(\kappa') \\
\mathcal{U}(\texttt{unit}) &= \langle\!\langle\rangle\!\rangle \\
\mathcal{U}(\texttt{dprod(}\kappa; u.\kappa'\texttt{)}) &= (\widehat{u} :: \mathcal{U}(\kappa)) \times \mathcal{U}(\kappa') \\
\mathcal{U}(\texttt{Type}) &= \texttt{T} \\
\mathcal{U}(\texttt{S(}\tau\texttt{)}) &= [{=}\mathcal{U}(\tau)]
\end{aligned}$$

- Each construction, c, except for constructions of the form $\texttt{con(}M\texttt{)}$ where M is not a module variable, maps onto an unexpanded type, $\mathcal{U}(c)$, as follows:

$$\begin{aligned}
\mathcal{U}(u) &= \widehat{u} \\
\mathcal{U}(\texttt{abs(}u.c\texttt{)}) &= \lambda\widehat{u}.\mathcal{U}(c) \\
\mathcal{U}(\texttt{app(}c;c'\texttt{)}) &= \mathcal{U}(c)(\mathcal{U}(c')) \\
\mathcal{U}(\texttt{triv}) &= \langle\!\langle\rangle\!\rangle \\
\mathcal{U}(\texttt{pair(}c;c'\texttt{)}) &= \langle\!\langle\mathcal{U}(c),\mathcal{U}(c')\rangle\!\rangle \\
\mathcal{U}(\texttt{prl(}c\texttt{)}) &= \mathcal{U}(c)\cdot\texttt{l} \\
\mathcal{U}(\texttt{prr(}c\texttt{)}) &= \mathcal{U}(c)\cdot\texttt{r} \\
\mathcal{U}(\texttt{parr(}\tau_1;\tau_2\texttt{)}) &= \mathcal{U}(\tau_1) \rightharpoonup \mathcal{U}(\tau_2) \\
\mathcal{U}(\texttt{all\{}\kappa\texttt{\}(}u.\tau\texttt{)}) &= \forall(\widehat{u} :: \mathcal{U}(\kappa)).\mathcal{U}(\tau) \\
\mathcal{U}(\texttt{rec(}t.\tau\texttt{)}) &= \mu\widehat{t}.\mathcal{U}(\tau) \\
\mathcal{U}(\texttt{prod}[L]\texttt{(}\{i \hookrightarrow \tau_i\}_{i\in L}\texttt{)}) &= \langle\{i \hookrightarrow \mathcal{U}(\tau_i)\}_{i\in L}\rangle \\
\mathcal{U}(\texttt{sum}[L]\texttt{(}\{i \hookrightarrow \tau_i\}_{i\in L}\texttt{)}) &= [\{i \hookrightarrow \mathcal{U}(\tau_i)\}_{i\in L}] \\
\mathcal{U}(\texttt{con(}X\texttt{)}) &= \widehat{X}\cdot\texttt{c}
\end{aligned}$$

- Each expression variable, x, maps onto a unique expression identifier, written $\widehat{x}$.
- Each expanded expression, e, except expressions of the form $\texttt{val(}M\texttt{)}$ where M is not a module variable, maps onto an unexpanded expression, $\mathcal{U}(e)$, as follows:

$$
\begin{aligned}
\mathcal{U}(x) &= \widehat{x} \\
\mathcal{U}(\texttt{lam\{}\tau\texttt{\}(}x.e\texttt{)}) &= \lambda\widehat{x}{:}\mathcal{U}(\tau).\mathcal{U}(e) \\
\mathcal{U}(\texttt{ap(}e_1;e_2\texttt{)}) &= \mathcal{U}(e_1)(\mathcal{U}(e_2)) \\
\mathcal{U}(\texttt{clam\{}\kappa\texttt{\}(}u.e\texttt{)}) &= \Lambda\widehat{u}{::}\mathcal{U}(\kappa).\mathcal{U}(e) \\
\mathcal{U}(\texttt{cap\{}c\texttt{\}(}e\texttt{)}) &= \mathcal{U}(e)\texttt{[}\mathcal{U}(c)\texttt{]} \\
\mathcal{U}(\texttt{fold(}e\texttt{)}) &= \texttt{fold}(\mathcal{U}(e)) \\
\mathcal{U}(\texttt{unfold(}e\texttt{)}) &= \texttt{unfold}(\mathcal{U}(e)) \\
\mathcal{U}(\texttt{tpl}[L]\texttt{(}\{i \hookrightarrow e_i\}_{i\in L}\texttt{)}) &= \langle\{i \hookrightarrow \mathcal{U}(e_i)\}_{i\in L}\rangle \\
\mathcal{U}(\texttt{prj}[\ell]\texttt{(}e\texttt{)}) &= \mathcal{U}(e)\cdot\ell \\
\mathcal{U}(\texttt{inj}[\ell]\texttt{(}e\texttt{)}) &= \texttt{inj}[\ell](\mathcal{U}(e)) \\
\mathcal{U}(\texttt{match}[n]\texttt{(}e;\{r_i\}_{1\le i\le n}\texttt{)}) &= \texttt{match}\,\mathcal{U}(e)\ \{\mathcal{U}(r_i)\}_{1\le i\le n} \\
\mathcal{U}(\texttt{val(}X\texttt{)}) &= \widehat{X}\cdot\texttt{v}
\end{aligned}
$$

- Each expanded rule, r, maps onto an unexpanded rule, $\mathcal{U}(r)$, as follows:

$$\mathcal{U}(\texttt{rule(}p.e\texttt{)}) = \texttt{urule(}\mathcal{U}(p).\mathcal{U}(e)\texttt{)}$$

- Each expanded pattern, p, maps onto an unexpanded pattern, $\mathcal{U}(p)$, as follows:

$$
\begin{aligned}
\mathcal{U}(x) &= \widehat{x} \\
\mathcal{U}(\texttt{wildp}) &= \texttt{uwildp} \\
\mathcal{U}(\texttt{foldp(}p\texttt{)}) &= \texttt{ufoldp(}\mathcal{U}(p)\texttt{)} \\
\mathcal{U}(\texttt{tplp}[L]\texttt{(}\{i \hookrightarrow p_i\}_{i\in L}\texttt{)}) &= \texttt{utplp}[L]\texttt{(}\{i \hookrightarrow \mathcal{U}(p_i)\}_{i\in L}\texttt{)} \\
\mathcal{U}(\texttt{injp}[\ell]\texttt{(}p\texttt{)}) &= \texttt{uinjp}[\ell]\texttt{(}\mathcal{U}(p)\texttt{)}
\end{aligned}
$$

Textual Syntax

There is also a context-free textual syntax for the UL. We need only posit the existence of partial metafunctions that satisfy the following condition.

Condition C.12 (Textual Representability)**.**

1. *For each $\hat{\kappa}$, there exists b such that* $\mathsf{parseUKind}(b) = \hat{\kappa}$.
2. *For each $\hat{c}$, there exists b such that* $\mathsf{parseUCon}(b) = \hat{c}$.
3. *For each $\hat{e}$, there exists b such that* $\mathsf{parseUExp}(b) = \hat{e}$.
4. *For each $\hat{p}$, there exists b such that* $\mathsf{parseUPat}(b) = \hat{p}$.

Condition C.13 (Expression Parsing Monotonicity)**.** *If* $\mathsf{parseUExp}(b) = \hat{e}$ *then* $\|\hat{e}\| < \|b\|$.

Condition C.14 (Pattern Parsing Monotonicity)**.** *If* $\mathsf{parseUPat}(b) = \hat{p}$ *then* $\|\hat{p}\| < \|b\|$.

C.2.2 Typed Expansion

Unexpanded Unified Contexts

A *unexpanded unified context*, $\hat{\Omega}$, takes the form $\langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle$, where $\mathcal{M}$ is a *module identifier expansion context*, $\mathcal{D}$ is a *construction identifier expansion context*, $\mathcal{G}$ is an *expression identifier expansion context*, and Ω is a unified context.

A module identifier expansion context, $\mathcal{M}$, is a finite function that maps each module identifier $\hat{X} \in \text{dom}(\mathcal{M})$ to the module identifier expansion $\hat{X} \rightsquigarrow X$. We write $\hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma$ when $\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle$ as an abbreviation of

$$\langle \mathcal{M} \uplus \hat{X} \rightsquigarrow X; \mathcal{D}; \mathcal{G}; \Omega, X : \sigma \rangle$$

A construction identifier expansion context, $\mathcal{D}$, is a finite function that maps each construction identifier $\hat{u} \in \text{dom}(\mathcal{D})$ to the construction identifier expansion $\hat{u} \rightsquigarrow u$. We write $\hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa$ when $\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle$ as an abbreviation of

$$\langle \mathcal{M}; \mathcal{D} \uplus \hat{u} \rightsquigarrow u; \mathcal{G}; \Omega, u :: \kappa \rangle$$

An expression identifier expansion context, $\mathcal{G}$, is a finite function that maps each expression identifier $\hat{x} \in \text{dom}(\mathcal{G})$ to the expression identifier expansion $\hat{x} \rightsquigarrow x$. We write $\hat{\Omega}, \hat{x} \rightsquigarrow x : \tau$ when $\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle$ as an abbreviation of

$$\langle \mathcal{M}; \mathcal{D}; \mathcal{G} \uplus \hat{x} \rightsquigarrow x; \Omega, x : \tau \rangle$$

Body Encoding and Decoding

An assumed 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* $b \downarrow_{\mathsf{Body}} e_{\text{body}}$. An inverse mapping is defined by the *body decoding judgement* $e_{\text{body}} \uparrow_{\mathsf{Body}} b$.

Judgement Form	**Description**
$b \downarrow_{\mathsf{Body}} e$	b has encoding e
$e \uparrow_{\mathsf{Body}} b$	e has decoding b

The following condition establishes an isomorphism between literal bodies and values of type `Body` mediated by the judgements above.

Condition C.15 (Body Isomorphism)**.**

1. *For every literal body b, we have that $b \downarrow_{\mathsf{Body}} e_{body}$ for some e_{body} such that $\vdash e_{body} : \mathtt{Body}$ and e_{body} val.*
2. *If $\vdash e_{body} : \mathtt{Body}$ and e_{body} val then $e_{body} \uparrow_{\mathsf{Body}} b$ for some b.*
3. *If $b \downarrow_{\mathsf{Body}} e_{body}$ then $e_{body} \uparrow_{\mathsf{Body}} b$.*
4. *If $\vdash e_{body} : \mathtt{Body}$ and e_{body} val and $e_{body} \uparrow_{\mathsf{Body}} b$ then $b \downarrow_{\mathsf{Body}} e_{body}$.*
5. *If $b \downarrow_{\mathsf{Body}} e_{body}$ and $b \downarrow_{\mathsf{Body}} e'_{body}$ then $e_{body} = e'_{body}$.*
6. *If $\vdash e_{body} : \mathtt{Body}$ and e_{body} val and $e_{body} \uparrow_{\mathsf{Body}} b$ and $e_{body} \uparrow_{\mathsf{Body}} b'$ then $b = b'$.*

We also assume a partial metafunction, $\mathsf{subseq}(b; m; n)$, which extracts a subsequence of b starting at position m and ending at position n, inclusive, where m and n are natural numbers. The following condition is technically necessary.

Condition C.16 (Body Subsequencing)**.** *If* $\mathsf{subseq}(b; m; n) = b'$ *then* $\|b'\| \leq \|b\|$.

Parse Results

The type function abbreviated `ParseResult`, and auxiliary abbreviations used below, is defined as follows:

$$L_{\mathtt{P}} \stackrel{\text{def}}{=} \mathtt{ParseError}, \mathtt{Success}$$

$$\mathtt{ParseResult} \stackrel{\text{def}}{=} \mathtt{abs}(t.\mathtt{sum}[L_{\mathtt{P}}](\mathtt{ParseError} \hookrightarrow \langle\rangle, \mathtt{Success} \hookrightarrow t))$$

$$\mathtt{ParseResult}(\tau) \stackrel{\text{def}}{=} \mathtt{app}(\mathtt{ParseResult}; \tau)$$

TSM Contexts

peTSM contexts, $\hat{\Psi}$, are of the form $\langle \mathcal{A}; \Psi \rangle$, where $\mathcal{A}$ is a *TSM identifier expansion context* and Ψ is a *peTSM definition context*.

ppTSM contexts, $\hat{\Phi}$, are of the form $\langle \mathcal{A}; \Phi \rangle$, where $\mathcal{A}$ is a TSM identifier expansion context and Φ is a *ppTSM definition context*.

A *TSM identifier expansion context*, $\mathcal{A}$, is a finite function mapping each TSM identifier $\hat{a} \in \mathrm{dom}(\mathcal{A})$ to the *TSM identifier expansion*, $\hat{a} \rightsquigarrow \epsilon$, for some *TSM expression*, ϵ. We write $\mathcal{A} \uplus \hat{a} \rightsquigarrow \epsilon$ for the TSM identifier expansion context that maps $\hat{a}$ to $\hat{a} \rightsquigarrow \epsilon$, and defers to $\mathcal{A}$ for all other TSM identifiers (i.e. the previous mapping is *updated*.)

A *peTSM definition context*, Ψ, is a finite function mapping each TSM name $a \in \mathrm{dom}(\Psi)$ to an *expanded peTSM definition*, $a \hookrightarrow \mathtt{petsm}(\rho; e_{\text{parse}})$, where ρ is the peTSM's type annotation, and e_{parse} is its parse function. We write $\Psi, a \hookrightarrow \mathtt{petsm}(\rho; e_{\text{parse}})$ when $a \notin \mathrm{dom}(\Psi)$ for the extension of Ψ that maps a to $a \hookrightarrow \mathtt{petsm}(\rho; e_{\text{parse}})$. We write $\Omega \vdash \Psi\ \mathsf{peTSMs}$ when all the TSM type annotations in Ψ are well-formed assuming Ω, and the parse functions in Ψ are closed and of the appropriate type.

Definition C.17 (peTSM Definition Context Formation)**.** $\Omega \vdash \Psi\ \mathsf{peTSMs}$ *iff for each* $a \hookrightarrow \mathtt{petsm}(\rho; e_{parse}) \in \Psi$*, we have* $\Omega \vdash \rho\ \mathsf{tsmty}$ *and*

$$\emptyset \vdash e_{parse} : \mathtt{parr}(\mathtt{Body}; \mathtt{ParseResult}(\mathtt{PPrExpr}))$$

Definition C.18 (peTSM Context Formation)**.** $\Omega \vdash \langle \mathcal{A}; \Psi \rangle\ \mathsf{peTSMctx}$ *iff* $\Omega \vdash \Psi\ \mathsf{peTSMs}$ *and for each* $\hat{a} \rightsquigarrow \epsilon \in \mathcal{A}$ *we have* $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon\ @\ \rho$ *for some* ρ.

A *ppTSM definition context*, Φ, is a finite function mapping each TSM name $a \in \mathrm{dom}(\Phi)$ to an *expanded ppTSM definition*, $a \hookrightarrow \mathtt{pptsm}(\rho; e_{\text{parse}})$, where ρ is the ppTSM's type annotation, and e_{parse} is its parse function. We write $\Phi, a \hookrightarrow \mathtt{pptsm}(\rho; e_{\text{parse}})$ when $a \notin \mathrm{dom}(\Phi)$ for the extension of Φ that maps a to $a \hookrightarrow \mathtt{pptsm}(\rho; e_{\text{parse}})$. We write $\Omega \vdash \Phi\ \mathsf{ppTSMs}$ when all the type annotations in Φ are well-formed assuming Ω, and the parse functions in Φ are closed and of the appropriate type.

Definition C.19 (ppTSM Definition Context Formation). $\Omega \vdash \Phi$ ppTSMs *iff for each* $\hat{a} \hookrightarrow \mathtt{pptsm}(\rho; e_{parse}) \in \Phi$, *we have* $\Omega \vdash \rho$ tsmty *and*

$$\emptyset \vdash e_{parse} : \mathtt{parr}(\mathtt{Body}; \mathtt{ParseResult}(\mathtt{PPrPat}))$$

Definition C.20 (ppTSM Context Formation). $\Omega \vdash \langle \mathcal{A}; \Phi \rangle$ ppTSMctx *iff* $\Omega \vdash \Phi$ ppTSMs *and for each* $\hat{a} \rightsquigarrow \epsilon \in \mathcal{A}$ *we have* $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mathbin{@} \rho$ *for some* ρ.

Signature and Module Expansion

$\boxed{\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma \text{ sig}}$ $\hat{\sigma}$ has well-formed expansion σ

$$\frac{\hat{\Omega} \vdash \hat{\kappa} \rightsquigarrow \kappa \text{ kind} \qquad \hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa \vdash \hat{\tau} \rightsquigarrow \tau :: \mathsf{Type}}{\hat{\Omega} \vdash [\![\hat{u} :: \hat{\kappa}; \hat{\tau}]\!] \rightsquigarrow \mathtt{sig}\{\kappa\}(u.\tau) \text{ sig}} \tag{C.15}$$

$\boxed{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma}$ $\hat{M}$ has expansion M matching σ

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma \qquad \hat{\Omega} \vdash \sigma <: \sigma'}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma'} \tag{C.16a}$$

$$\frac{}{\hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma \vdash_{\hat{\Psi};\hat{\Phi}} \hat{X} \rightsquigarrow X : \sigma} \tag{C.16b}$$

$$\frac{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c :: \kappa \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : [c/u]\tau}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} [\![\hat{c}; \hat{e}]\!] \rightsquigarrow \mathtt{struct}(c; e) : \mathtt{sig}\{\kappa\}(u.\tau)} \tag{C.16c}$$

$$\frac{\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma \text{ sig} \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{M} \upharpoonright \hat{\sigma} \rightsquigarrow \mathtt{seal}\{\sigma\}(M) : \sigma} \tag{C.16d}$$

$$\frac{\begin{array}{c}\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma \qquad \hat{\Omega} \vdash \hat{\sigma}' \rightsquigarrow \sigma' \text{ sig} \\ \hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma \vdash_{\hat{\Psi};\hat{\Phi}} \hat{M}' \rightsquigarrow M' : \sigma'\end{array}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} (\mathtt{let}\ \hat{X} = \hat{M}\ \mathtt{in}\ \hat{M}') : \hat{\sigma}' \rightsquigarrow \mathtt{mlet}\{\sigma'\}(M; X.M') : \sigma'} \tag{C.16e}$$

$$\frac{\begin{array}{c}\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho \text{ tsmty} \qquad \emptyset \vdash e_{\text{parse}} : \mathtt{parr}(\mathtt{Body}; \mathtt{ParseResult}(\mathtt{PPrExpr})) \\ e_{\text{parse}} \Downarrow e'_{\text{parse}} \qquad \hat{\Omega} \vdash_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \mathtt{defref}[a]; \Psi, a \hookrightarrow \mathtt{petsm}(\rho; e'_{\text{parse}}) \rangle; \hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma\end{array}}{\hat{\Omega} \vdash_{\langle \mathcal{A}; \Psi \rangle; \hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for\ expressions\ by\ static}\ e_{\text{parse}}\ \mathtt{in}\ \hat{M} \rightsquigarrow M : \sigma} \tag{C.16f}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\langle \mathcal{A}; \Psi \rangle} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \rho \qquad \hat{\Omega} \vdash_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \epsilon_{\text{normal}}; \Psi \rangle; \hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma}{\hat{\Omega} \vdash_{\langle \mathcal{A}; \Psi \rangle; \hat{\Phi}} \mathtt{let\ syntax}\ \hat{a} = \hat{\epsilon}\ \mathtt{for\ expressions\ in}\ \hat{M} \rightsquigarrow M : \sigma} \tag{C.16g}$$

$$\frac{\begin{array}{c}\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho \text{ tsmty} \qquad \emptyset \vdash e_{\text{parse}} : \mathtt{parr}(\mathtt{Body}; \mathtt{ParseResult}(\mathtt{PPrPat})) \\ e_{\text{parse}} \Downarrow e'_{\text{parse}} \qquad \hat{\Omega} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \hookrightarrow \mathtt{defref}[a]; \Phi, a \hookrightarrow \mathtt{pptsm}(\rho; e'_{\text{parse}}) \rangle} \hat{M} \rightsquigarrow M : \sigma\end{array}}{\hat{\Omega} \vdash_{\hat{\Psi}; \langle \mathcal{A}; \Phi \rangle} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for\ patterns\ by\ static}\ e_{\text{parse}}\ \mathtt{in}\ \hat{M} \rightsquigarrow M : \sigma} \tag{C.16h}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\langle \mathcal{A}; \Phi \rangle} \hat{\epsilon} \leadsto \epsilon \mathbin{@} \rho \qquad \hat{\Omega} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \hookrightarrow \epsilon; \Phi \rangle} \hat{M} \leadsto M : \sigma}{\hat{\Omega} \vdash_{\hat{\Psi}; \langle \mathcal{A}; \Phi \rangle} \texttt{let syntax } \hat{a} = \hat{\epsilon} \texttt{ for patterns in } \hat{M} \leadsto M : \sigma} \tag{C.16i}$$

Kind and Construction Expansion

$\boxed{\hat{\Omega} \vdash \hat{\kappa} \leadsto \kappa \ \mathsf{kind}}$ $\hat{\kappa}$ has well-formed expansion κ

$$\frac{\hat{\Omega} \vdash \hat{\kappa}_1 \leadsto \kappa_1 \ \mathsf{kind} \qquad \hat{\Omega}, \hat{u} \leadsto u :: \kappa_1 \vdash \hat{\kappa}_2 \leadsto \kappa_2 \ \mathsf{kind}}{\hat{\Omega} \vdash (\hat{u} :: \hat{\kappa}_1) \to \hat{\kappa}_2 \leadsto \texttt{darr}(\kappa_1; u.\kappa_2) \ \mathsf{kind}} \tag{C.17a}$$

$$\frac{}{\hat{\Omega} \vdash \langle\!\langle \rangle\!\rangle \leadsto \texttt{unit} \ \mathsf{kind}} \tag{C.17b}$$

$$\frac{\hat{\Omega} \vdash \hat{\kappa}_1 \leadsto \kappa_1 \ \mathsf{kind} \qquad \hat{\Omega}, \hat{u} \leadsto u :: \kappa_1 \vdash \hat{\kappa}_2 \leadsto \kappa_2 \ \mathsf{kind}}{\hat{\Omega} \vdash (\hat{u} :: \hat{\kappa}_1) \times \hat{\kappa}_2 \leadsto \texttt{dprod}(\kappa_1; u.\kappa_2) \ \mathsf{kind}} \tag{C.17c}$$

$$\frac{}{\hat{\Omega} \vdash \texttt{T} \leadsto \texttt{Type} \ \mathsf{kind}} \tag{C.17d}$$

$$\frac{\hat{\Omega} \vdash \hat{\tau} \leadsto \tau :: \texttt{Type}}{\hat{\Omega} \vdash [=\hat{\tau}] \leadsto \texttt{S}(\tau) \ \mathsf{kind}} \tag{C.17e}$$

$\boxed{\hat{\Omega} \vdash \hat{c} \leadsto c :: \kappa}$ $\hat{c}$ has expansion c of kind κ

$$\frac{\hat{\Omega} \vdash \hat{c} \leadsto c :: \kappa_1 \qquad \Omega \vdash \kappa_1 <:: \kappa_2}{\hat{\Omega} \vdash \hat{c} \leadsto c :: \kappa_2} \tag{C.18a}$$

$$\frac{}{\hat{\Omega}, \hat{u} \leadsto u :: \kappa \vdash \hat{u} \leadsto u :: \kappa} \tag{C.18b}$$

$$\frac{\hat{\Omega}, \hat{u} \leadsto u :: \kappa_1 \vdash \hat{c}_2 \leadsto c_2 :: \kappa_2}{\hat{\Omega} \vdash \lambda \hat{u}.\hat{c}_2 \leadsto \texttt{abs}(u.c_2) :: \texttt{darr}(\kappa_1; u.\kappa_2)} \tag{C.18c}$$

$$\frac{\hat{\Omega} \vdash \hat{c}_1 \leadsto c_1 :: \texttt{darr}(\kappa_2; u.\kappa) \qquad \hat{\Omega} \vdash \hat{c}_2 \leadsto c_2 :: \kappa_2}{\hat{\Omega} \vdash \hat{c}_1(\hat{c}_2) \leadsto \texttt{app}(c_1; c_2) :: [c_1/u]\kappa} \tag{C.18d}$$

$$\frac{}{\hat{\Omega} \vdash \langle\!\langle \rangle\!\rangle \leadsto \texttt{triv} :: \texttt{unit}} \tag{C.18e}$$

$$\frac{\hat{\Omega} \vdash \hat{c}_1 \leadsto c_1 :: \kappa_1 \qquad \hat{\Omega} \vdash \hat{c}_2 \leadsto c_2 :: [c_1/u]\kappa_2}{\hat{\Omega} \vdash \langle\!\langle \hat{c}_1, \hat{c}_2 \rangle\!\rangle \leadsto \texttt{pair}(c_1; c_2) :: \texttt{dprod}(\kappa_1; u.\kappa_2)} \tag{C.18f}$$

$$\frac{\hat{\Omega} \vdash \hat{c} \leadsto c :: \texttt{dprod}(\kappa_1; u.\kappa_2)}{\hat{\Omega} \vdash \hat{c} \cdot \texttt{l} \leadsto \texttt{prl}(c) :: \kappa_1} \tag{C.18g}$$

$$\frac{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c :: \mathtt{dprod}(\kappa_1; u.\kappa_2)}{\hat{\Omega} \vdash \hat{c} \cdot \mathtt{r} \rightsquigarrow \mathtt{prr}(c) :: [\mathtt{prl}(c)/u]\kappa_2} \tag{C.18h}$$

$$\frac{\hat{\Omega} \vdash \hat{\tau}_1 \rightsquigarrow \tau_1 :: \mathsf{Type} \qquad \hat{\Omega} \vdash \hat{\tau}_2 \rightsquigarrow \tau_2 :: \mathsf{Type}}{\hat{\Omega} \vdash \hat{\tau}_1 \rightharpoonup \hat{\tau}_2 \rightsquigarrow \mathtt{parr}(\tau_1; \tau_2) :: \mathsf{Type}} \tag{C.18i}$$

$$\frac{\hat{\Omega} \vdash \hat{\kappa} \rightsquigarrow \kappa \ \mathsf{kind} \qquad \hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa \vdash \hat{\tau} \rightsquigarrow \tau :: \mathsf{Type}}{\hat{\Omega} \vdash \forall(\hat{u} :: \hat{\kappa}).\hat{\tau} \rightsquigarrow \mathtt{all}\{\kappa\}(u.\tau) :: \mathsf{Type}} \tag{C.18j}$$

$$\frac{\hat{\Omega}, \hat{t} \rightsquigarrow t :: \mathsf{Type} \vdash \hat{\tau} \rightsquigarrow \tau :: \mathsf{Type}}{\hat{\Omega} \vdash \mu\hat{t}.\hat{\tau} \rightsquigarrow \mathtt{rec}(t.\tau) :: \mathsf{Type}} \tag{C.18k}$$

$$\frac{\{\hat{\Omega} \vdash \hat{\tau}_i \rightsquigarrow \tau_i :: \mathsf{Type}\}_{1 \le i \le n}}{\hat{\Omega} \vdash \langle \{i \hookrightarrow \hat{\tau}_i\}_{i \in L} \rangle \rightsquigarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) :: \mathsf{Type}} \tag{C.18l}$$

$$\frac{\{\hat{\Omega} \vdash \hat{\tau}_i \rightsquigarrow \tau_i :: \mathsf{Type}\}_{1 \le i \le n}}{\hat{\Omega} \vdash [\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}] \rightsquigarrow \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) :: \mathsf{Type}} \tag{C.18m}$$

$$\frac{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c :: \mathsf{Type}}{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c :: \mathsf{S}(c)} \tag{C.18n}$$

$$\frac{}{\hat{\Omega}, \hat{X} \rightsquigarrow X : \mathtt{sig}\{\kappa\}(u.\tau) \vdash \hat{X} \cdot \mathtt{c} \rightsquigarrow \mathtt{con}(X) :: \kappa} \tag{C.18o}$$

Type, Expression, Rule and Pattern Expansion

$\boxed{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e : \tau}$ $\hat{e}$ has expansion e of type τ

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e : \tau \qquad \Omega \vdash \tau <: \tau'}{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e : \tau'} \tag{C.19a}$$

$$\frac{}{\hat{\Omega}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{x} \rightsquigarrow x : \tau} \tag{C.19b}$$

$$\frac{\hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau :: \mathsf{Type} \qquad \hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e : \tau}{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} : \hat{\tau} \rightsquigarrow e : \tau} \tag{C.19c}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e}_1 \rightsquigarrow e_1 : \tau_1 \qquad \hat{\Omega}, \hat{x} \rightsquigarrow x : \tau_1 \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e}_2 \rightsquigarrow e_2 : \tau_2}{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \mathtt{let\,val}\ \hat{x} = \hat{e}_1\ \mathtt{in}\ \hat{e}_2 \rightsquigarrow \mathtt{ap}(\mathtt{lam}\{\tau_1\}(x.e_2); e_1) : \tau_2} \tag{C.19d}$$

$$\frac{\hat{\Omega} \vdash \hat{\tau}_1 \rightsquigarrow \tau_1 :: \mathsf{Type} \qquad \hat{\Omega}, \hat{x} \rightsquigarrow x : \tau_1 \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e : \tau_2}{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \lambda\hat{x}{:}\hat{\tau}_1.\hat{e} \rightsquigarrow \mathtt{lam}\{\tau_1\}(x.e) : \mathtt{parr}(\tau_1; \tau_2)} \tag{C.19e}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_1 \leadsto e_1 : \texttt{parr}(\tau_2; \tau) \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_2 \leadsto e_2 : \tau_2}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_1(\hat{e}_2) \leadsto \texttt{ap}(e_1; e_2) : \tau} \tag{C.19f}$$

$$\frac{\hat{\Omega} \vdash \hat{\kappa} \leadsto \kappa \text{ kind} \qquad \hat{\Omega}, \hat{u} \leadsto u :: \kappa \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e : \tau}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \Lambda \hat{u}{::}\hat{\kappa}.\hat{e} \leadsto \texttt{clam}\{\kappa\}(u.e) : \texttt{all}\{\kappa\}(u.\tau)} \tag{C.19g}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e : \texttt{all}\{\kappa\}(u.\tau) \qquad \hat{\Omega} \vdash \hat{c} \leadsto c \Rightarrow \kappa}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}[\hat{c}] \leadsto \texttt{cap}\{c\}(e) : [c/t]\tau} \tag{C.19h}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e : [\texttt{rec}(t.\tau)/t]\tau}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{fold}(\hat{e}) \leadsto \texttt{fold}(e) : \texttt{rec}(t.\tau)} \tag{C.19i}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e : \texttt{rec}(t.\tau)}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{unfold}(\hat{e}) \leadsto \texttt{unfold}(e) : [\texttt{rec}(t.\tau)/t]\tau} \tag{C.19j}$$

$$\frac{\{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_i \leadsto e_i : \tau_i\}_{i \in L}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \langle\{i \hookrightarrow \hat{e}_i\}_{i \in L}\rangle \leadsto \texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{C.19k}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e : \texttt{prod}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \cdot \ell \leadsto \texttt{prj}[\ell](e) : \tau} \tag{C.19l}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \leadsto e' : \tau'}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{inj}[\ell](\hat{e}) \leadsto \texttt{inj}[\ell](e') : \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau')} \tag{C.19m}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e : \tau \qquad \{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \leadsto r_i \Rightarrow \tau \Mapsto \tau'\}_{1 \le i \le n}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{match}\, \hat{e}\, \{\hat{r}_i\}_{1 \le i \le n} \leadsto \texttt{match}[n](e; \{r_i\}_{1 \le i \le n}) : \tau'} \tag{C.19n}$$

$$\frac{}{\hat{\Omega}, \hat{X} \leadsto X : \texttt{sig}\{\kappa\}(u.\tau) \vdash_{\hat{\Psi};\hat{\Phi}} \hat{X} \cdot \mathtt{v} \leadsto \texttt{val}(X) : [\texttt{con}(X)/u]\tau} \tag{C.19o}$$

$$\frac{\begin{array}{c} \hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\text{app}} \rangle \qquad \hat{\Psi} = \langle \mathcal{A}; \Psi \rangle \\ \hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{e} \leadsto \epsilon \mathbin{@} \texttt{type}(\tau_{\text{final}}) \qquad \Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \Downarrow \epsilon_{\text{normal}} \\ \mathsf{tsmdef}(\epsilon_{\text{normal}}) = a \qquad \Psi = \Psi', a \hookrightarrow \texttt{petsm}(\rho; e_{\text{parse}}) \\ b \downarrow_{\mathsf{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}[\texttt{SuccessE}](e_{\text{pproto}}) \qquad e_{\text{pproto}} \uparrow_{\mathsf{PPrExpr}} \dot{e} \\ \Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \dot{e} \looparrowright_{\epsilon_{\text{normal}}} \grave{e} \mathbin{?} \texttt{type}(\tau_{\text{proto}}) \dashv \omega : \Omega_{\text{params}} \\ \mathsf{seg}(\grave{e}) \text{ segments } b \qquad \Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}}; \hat{\Omega}; \hat{\Psi}; \hat{\Phi}; b} \grave{e} \leadsto e : \tau_{\text{proto}} \end{array}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}\ \text{`}b\text{`} \leadsto [\omega]e : [\omega]\tau_{\text{proto}}} \tag{C.19p}$$

$\boxed{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \rightsquigarrow r : \tau \Mapsto \tau'}$ $\hat{r}$ has expansion r taking values of type τ to values of type τ'

$$\frac{\begin{array}{c}\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle \\ \hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \langle \emptyset; \emptyset; \mathcal{G}'; \Omega' \rangle \qquad \langle \mathcal{M}; \mathcal{D}; \mathcal{G} \uplus \mathcal{G}'; \Omega \cup \Omega' \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau'\end{array}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{p} \Rightarrow \hat{e} \rightsquigarrow \texttt{rule}(p.e) : \tau \Mapsto \tau'} \tag{C.20}$$

$\boxed{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \hat{\Omega}'}$ $\hat{p}$ has expansion p matching against τ generating hypotheses $\hat{\Omega}'$

$$\frac{\begin{array}{c}\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle \\ \hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \hat{\Omega}' \qquad \Omega \vdash \tau <: \tau'\end{array}}{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau' \dashv\!\mid \hat{\Omega}'} \tag{C.21a}$$

$$\frac{}{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{x} \rightsquigarrow x : \tau \dashv\!\mid \langle \emptyset; \emptyset; \hat{x} \rightsquigarrow x; x : \tau \rangle} \tag{C.21b}$$

$$\frac{}{\hat{\Omega} \vdash_{\hat{\Phi}} \_ \rightsquigarrow \texttt{wildp} : \tau \dashv\!\mid \langle \emptyset; \emptyset; \emptyset; \emptyset \rangle} \tag{C.21c}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : [\texttt{rec}(t.\tau)/t]\tau \dashv\!\mid \hat{\Omega}'}{\hat{\Omega} \vdash_{\hat{\Phi}} \texttt{fold}(\hat{p}) \rightsquigarrow \texttt{foldp}(p) : \texttt{rec}(t.\tau) \dashv\!\mid \hat{\Omega}'} \tag{C.21d}$$

$$\frac{\begin{array}{c}\tau = \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \\ \{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{p}_i \rightsquigarrow p_i : \tau_i \dashv\!\mid \hat{\Omega}_i\}_{i \in L}\end{array}}{\hat{\Omega} \vdash_{\hat{\Phi}} \langle \{i \hookrightarrow \hat{p}_i\}_{i \in L} \rangle \rightsquigarrow \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \tau \dashv\!\mid \cup_{i \in L} \hat{\Omega}_i} \tag{C.21e}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \hat{\Omega}'}{\hat{\Omega} \vdash_{\hat{\Phi}} \texttt{inj}[\ell](\hat{p}) \rightsquigarrow \texttt{injp}[\ell](p) : \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\!\mid \hat{\Omega}'} \tag{C.21f}$$

$$\frac{\begin{array}{c}\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\text{app}} \rangle \qquad \hat{\Phi} = \langle \mathcal{A}; \Phi \rangle \\ \hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \hat{\epsilon} \rightsquigarrow \epsilon \mathrel{@} \texttt{type}(\tau_{\text{final}}) \qquad \Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \Downarrow \epsilon_{\text{normal}} \\ \mathsf{tsmdef}(\epsilon_{\text{normal}}) = a \qquad \Phi = \Phi', a \hookrightarrow \texttt{pptsm}(\rho; e_{\text{parse}}) \\ b \downarrow_{\mathsf{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}[\mathsf{SuccessP}](e_{\text{pproto}}) \qquad e_{\text{pproto}} \uparrow_{\mathsf{PPrPat}} \grave{p} \\ \Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi} \grave{p} \looparrowright_{\epsilon_{\text{normal}}} \grave{p} \mathrel{?} \texttt{type}(\tau_{\text{proto}}) \dashv \omega : \Omega_{\text{params}} \\ \mathsf{seg}(\grave{p}) \text{ segments } b \qquad \grave{p} \rightsquigarrow p : \tau_{\text{proto}} \dashv\!\mid^{\omega:\Omega_{\text{params}}; \hat{\Omega}; \hat{\Phi}; b} \hat{\Omega}'\end{array}}{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{\epsilon} \text{ `}b\text{` } \rightsquigarrow p : [\omega]\tau_{\text{proto}} \dashv\!\mid \hat{\Omega}'} \tag{C.21g}$$

TSM Type and Expression Expansion

$\boxed{\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho \ \mathsf{tsmty}}$ $\hat{\rho}$ has well-formed expansion ρ

$$\frac{\hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau :: \mathtt{Type}}{\hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \texttt{type}(\tau) \ \mathsf{tsmty}} \tag{C.22a}$$

$$\frac{\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma \;\mathsf{sig} \qquad \hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma \vdash \hat{\rho} \rightsquigarrow \rho \;\mathsf{tsmty}}{\hat{\Omega} \vdash \forall \hat{X}{:}\hat{\sigma}.\hat{\rho} \rightsquigarrow \texttt{allmods}\{\sigma\}(X.\rho) \;\mathsf{tsmty}} \tag{C.22b}$$

$\boxed{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{\epsilon} \rightsquigarrow \epsilon @ \rho}$ $\hat{\epsilon}$ has peTSM expression expansion ϵ at ρ

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon @ \rho}{\langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle \vdash^{\mathsf{Exp}}_{\langle \mathcal{A}, \hat{a} \hookrightarrow \epsilon; \Psi \rangle} \hat{a} \rightsquigarrow \epsilon @ \rho} \tag{C.23a}$$

$$\frac{\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma \;\mathsf{sig} \qquad \hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{\epsilon} \rightsquigarrow \epsilon @ \rho}{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \Lambda \hat{X}{:}\hat{\sigma}.\hat{\epsilon} \rightsquigarrow \texttt{absmod}\{\sigma\}(X.\epsilon) @ \texttt{allmods}\{\sigma\}(X.\rho)} \tag{C.23b}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{\epsilon} \rightsquigarrow \epsilon @ \texttt{allmods}\{\sigma\}(X'.\rho) \qquad \hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{X} \rightsquigarrow X : \sigma}{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{\epsilon}(\hat{X}) \rightsquigarrow \texttt{apmod}\{X\}(\epsilon) @ [X/X']\rho} \tag{C.23c}$$

$\boxed{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Psi}} \hat{\epsilon} \rightsquigarrow \epsilon @ \rho}$ $\hat{\epsilon}$ has ppTSM expression expansion ϵ at ρ

$$\frac{\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon @ \rho}{\langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle \vdash^{\mathsf{Pat}}_{\langle \mathcal{A}, \hat{a} \hookrightarrow \epsilon; \Phi \rangle} \hat{a} \rightsquigarrow \epsilon @ \rho} \tag{C.24a}$$

$$\frac{\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma \;\mathsf{sig} \qquad \hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \hat{\epsilon} \rightsquigarrow \epsilon @ \rho}{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \Lambda \hat{X}{:}\hat{\sigma}.\hat{\epsilon} \rightsquigarrow \texttt{absmod}\{\sigma\}(X.\epsilon) @ \texttt{allmods}\{\sigma\}(X.\rho)} \tag{C.24b}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \hat{\epsilon} \rightsquigarrow \epsilon @ \texttt{allmods}\{\sigma\}(X'.\rho) \qquad \hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{X} \rightsquigarrow X : \sigma}{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \hat{\epsilon}(\hat{X}) \rightsquigarrow \texttt{apmod}\{X\}(\epsilon) @ [X/X']\rho} \tag{C.24c}$$

Statics of the TSM Language

$\boxed{\Omega \vdash \rho \;\mathsf{tsmty}}$ ρ is a TSM type

$$\frac{\Omega \vdash \tau :: \texttt{Type}}{\Omega \vdash \texttt{type}(\tau) \;\mathsf{tsmty}} \tag{C.25a}$$

$$\frac{\Omega \vdash \sigma \;\mathsf{sig} \qquad \Omega, X : \sigma \vdash \rho \;\mathsf{tsmty}}{\Omega \vdash \texttt{allmods}\{\sigma\}(X.\rho) \;\mathsf{tsmty}} \tag{C.25b}$$

$\boxed{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon @ \rho}$ ϵ is a peTSM expression at ρ

$$\frac{\Omega \vdash \rho \;\mathsf{tsmty}}{\Omega \vdash^{\mathsf{Exp}}_{\Psi, a \hookrightarrow \texttt{petsm}(\rho; e_{\text{parse}})} \texttt{defref}[a] @ \rho} \tag{C.26a}$$

$$\frac{\Omega \vdash \sigma \,\mathsf{sig} \qquad \Omega, X : \sigma \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \,@\, \rho}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{absmod\{}\sigma\mathtt{\}(}X.\epsilon\mathtt{)} \,@\, \mathtt{allmods\{}\sigma\mathtt{\}(}X.\rho\mathtt{)}} \tag{C.26b}$$

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \,@\, \mathtt{allmods\{}\sigma\mathtt{\}(}X'.\rho\mathtt{)} \qquad \Omega \vdash X : \sigma}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{apmod\{}X\mathtt{\}(}\epsilon\mathtt{)} \,@\, [X/X']\rho} \tag{C.26c}$$

$\boxed{\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \,@\, \rho}$ ϵ is a ppTSM expression at ρ

$$\frac{\Omega \vdash \rho \,\mathsf{tsmty}}{\Omega \vdash^{\mathsf{Pat}}_{\Phi, a \hookrightarrow \mathtt{pptsm}(\rho; e_{\text{parse}})} \mathtt{defref[}a\mathtt{]} \,@\, \rho} \tag{C.27a}$$

$$\frac{\Omega \vdash \sigma \,\mathsf{sig} \qquad \Omega, X : \sigma \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \,@\, \rho}{\Omega \vdash^{\mathsf{Pat}}_{\Phi} \mathtt{absmod\{}\sigma\mathtt{\}(}X.\epsilon\mathtt{)} \,@\, \mathtt{allmods\{}\sigma\mathtt{\}(}X.\rho\mathtt{)}} \tag{C.27b}$$

$$\frac{\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \,@\, \mathtt{allmods\{}\sigma\mathtt{\}(}X'.\rho\mathtt{)} \qquad \Omega \vdash X : \sigma}{\Omega \vdash^{\mathsf{Pat}}_{\Phi} \mathtt{apmod\{}X\mathtt{\}(}\epsilon\mathtt{)} \,@\, [X/X']\rho} \tag{C.27c}$$

The following metafunction extracts the TSM name from a TSM expression.

$$\mathsf{tsmdef}(\mathtt{defref[}a\mathtt{]}) = a \tag{C.28a}$$

$$\mathsf{tsmdef}(\mathtt{absmod\{}\sigma\mathtt{\}(}X.\epsilon\mathtt{)}) = \mathsf{tsmdef}(\epsilon) \tag{C.28b}$$

$$\mathsf{tsmdef}(\mathtt{apmod\{}X\mathtt{\}(}\epsilon\mathtt{)}) = \mathsf{tsmdef}(\epsilon) \tag{C.28c}$$

Dynamics of the TSM Language

$\boxed{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto \epsilon'}$ peTSM expression ϵ transitions to ϵ'

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto \epsilon'}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{apmod\{}X\mathtt{\}(}\epsilon\mathtt{)} \mapsto \mathtt{apmod\{}X\mathtt{\}(}\epsilon'\mathtt{)}} \tag{C.29a}$$

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{apmod\{}X\mathtt{\}(absmod\{}\sigma\mathtt{\}(}X'.\epsilon\mathtt{))} \mapsto [X/X']\epsilon} \tag{C.29b}$$

$\boxed{\Omega \vdash^{\mathsf{Pat}}_{\Psi} \epsilon \mapsto \epsilon'}$ ppTSM expression ϵ transitions to ϵ'

$$\frac{\Omega \vdash^{\mathsf{Pat}}_{\Psi} \epsilon \mapsto \epsilon'}{\Omega \vdash^{\mathsf{Pat}}_{\Psi} \mathtt{apmod\{}X\mathtt{\}(}\epsilon\mathtt{)} \mapsto \mathtt{apmod\{}X\mathtt{\}(}\epsilon'\mathtt{)}} \tag{C.30a}$$

$$\frac{}{\Omega \vdash^{\mathsf{Pat}}_{\Psi} \mathtt{apmod\{}X\mathtt{\}(absmod\{}\sigma\mathtt{\}(}X'.\epsilon\mathtt{))} \mapsto [X/X']\epsilon} \tag{C.30b}$$

$\boxed{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto^{*} \epsilon'}$ peTSM expression ϵ transitions in multiple steps to ϵ'

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto^{*} \epsilon} \tag{C.31a}$$

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto \epsilon'}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto^* \epsilon'} \tag{C.31b}$$

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto^* \epsilon' \qquad \Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon' \mapsto^* \epsilon''}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto^* \epsilon''} \tag{C.31c}$$

$\boxed{\Omega \vdash^{\mathsf{Pat}}_{\Psi} \epsilon \mapsto^* \epsilon'}$ ppTSM expression ϵ transitions in multiple steps to ϵ'

$$\frac{}{\Omega \vdash^{\mathsf{Pat}}_{\Psi} \epsilon \mapsto^* \epsilon} \tag{C.32a}$$

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto \epsilon'}{\Omega \vdash^{\mathsf{Pat}}_{\Psi} \epsilon \mapsto^* \epsilon'} \tag{C.32b}$$

$$\frac{\Omega \vdash^{\mathsf{Pat}}_{\Psi} \epsilon \mapsto^* \epsilon' \qquad \Omega \vdash^{\mathsf{Pat}}_{\Psi} \epsilon' \mapsto^* \epsilon''}{\Omega \vdash^{\mathsf{Pat}}_{\Psi} \epsilon \mapsto^* \epsilon''} \tag{C.32c}$$

$\boxed{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \Downarrow \epsilon'}$ peTSM expression ϵ normalizes to ϵ'

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto^* \epsilon' \qquad \Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon' \;\mathsf{normal}}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \Downarrow \epsilon'} \tag{C.33}$$

$\boxed{\Omega \vdash^{\mathsf{Pat}}_{\Psi} \epsilon \Downarrow \epsilon'}$ ppTSM expression ϵ normalizes to ϵ'

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto^* \epsilon' \qquad \Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon' \;\mathsf{normal}}{\Omega \vdash^{\mathsf{Pat}}_{\Psi} \epsilon \Downarrow \epsilon'} \tag{C.34}$$

$\boxed{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \;\mathsf{normal}}$ ϵ is a normal peTSM expression

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi, a \hookrightarrow \mathtt{petsm}(\rho; e_{\text{parse}})} \mathtt{defref[}a\mathtt{]} \;\mathsf{normal}} \tag{C.35a}$$

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{absmod\{}\sigma\mathtt{\}(}X.\epsilon\mathtt{)} \;\mathsf{normal}} \tag{C.35b}$$

$$\frac{\epsilon \neq \mathtt{absmod\{}\sigma\mathtt{\}(}X'.\epsilon'\mathtt{)} \qquad \Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \;\mathsf{normal}}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{apmod\{}X\mathtt{\}(}\epsilon\mathtt{)} \;\mathsf{normal}} \tag{C.35c}$$

$\boxed{\Omega \vdash^{\mathsf{Pat}}_{\Psi} \epsilon \;\mathsf{normal}}$ ϵ is a normal ppTSM expression

$$\frac{}{\Omega \vdash^{\mathsf{Pat}}_{\Psi, a \hookrightarrow \mathtt{petsm}(\rho; e_{\text{parse}})} \mathtt{defref[}a\mathtt{]} \;\mathsf{normal}} \tag{C.36a}$$

$$\frac{}{\Omega \vdash^{\mathsf{Pat}}_{\Psi} \mathtt{absmod\{}\sigma\mathtt{\}(}X.\epsilon\mathtt{)} \;\mathsf{normal}} \tag{C.36b}$$

$$\frac{\epsilon \neq \mathtt{absmod\{}\sigma\mathtt{\}(}X'.\epsilon'\mathtt{)} \qquad \Omega \vdash^{\mathsf{Pat}}_{\Psi} \epsilon \;\mathsf{normal}}{\Omega \vdash^{\mathsf{Pat}}_{\Psi} \mathtt{apmod\{}X\mathtt{\}(}\epsilon\mathtt{)} \;\mathsf{normal}} \tag{C.36c}$$

C.3 Proto-Expansion Validation

C.3.1 Syntax of Proto-Expansions

Syntax – Parameterized Proto-Expressions

Sort			Operational Form	Stylized Form	Description
PPrExpr	$\dot{e}$	$::=$	$\texttt{prexp}(\grave{e})$	$\grave{e}$	proto-expression
			$\texttt{prbindmod}(X.\dot{e})$	$\Lambda X.\dot{e}$	module binding

Syntax – Parameterized Proto-Patterns

Sort			Operational Form	Stylized Form	Description
PPrPat	$\dot{p}$	$::=$	$\texttt{prpat}(\grave{p})$	$\grave{p}$	proto-pattern
			$\texttt{prbindmod}(X.\dot{p})$	$\Lambda X.\dot{p}$	module binding

Syntax – Proto-Kinds and Proto-Constructions

Sort			Operational Form	Stylized Form	Description
PrKind	$\grave{\kappa}$	$::=$	$\texttt{prdarr}(\grave{\kappa}; u.\grave{\kappa})$	$(u :: \grave{\kappa}) \to \grave{\kappa}$	dependent function
			$\texttt{prunit}$	$\langle\langle\rangle\rangle$	nullary product
			$\texttt{prdprod}(\grave{\kappa}; u.\grave{\kappa})$	$(u :: \grave{\kappa}) \times \grave{\kappa}$	dependent product
			$\texttt{prType}$	$\texttt{T}$	type
			$\texttt{prS}(\grave{\tau})$	$[=\grave{\tau}]$	singleton
			$\texttt{splicedk}[m; n]$	$\texttt{splicedk}[m; n]$	spliced kind
PrCon	$\grave{c}, \grave{\tau}$	$::=$	u	u	construction variable
			t	t	type variable
			$\texttt{prabs}(u.\grave{c})$	$\lambda u.\grave{c}$	abstraction
			$\texttt{prapp}(\grave{c}; \grave{c})$	$\grave{c}(\grave{c})$	application
			$\texttt{prtriv}$	$\langle\langle\rangle\rangle$	trivial
			$\texttt{prpair}(\grave{c}; \grave{c})$	$\langle\langle \grave{c}, \grave{c} \rangle\rangle$	pair
			$\texttt{prprl}(\grave{c})$	$\grave{c} \cdot \texttt{l}$	left projection
			$\texttt{prprr}(\grave{c})$	$\grave{c} \cdot \texttt{r}$	right projection
			$\texttt{prparr}(\grave{\tau}; \grave{\tau})$	$\grave{\tau} \rightharpoonup \grave{\tau}$	partial function
			$\texttt{prall}\{\grave{\kappa}\}(u.\grave{\tau})$	$\forall(u :: \grave{\kappa}).\grave{\tau}$	polymorphic
			$\texttt{prrec}(t.\grave{\tau})$	$\mu t.\grave{\tau}$	recursive
			$\texttt{prprod}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L})$	$\langle\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}\rangle$	labeled product
			$\texttt{prsum}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L})$	$[\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}]$	labeled sum
			$\texttt{prcon}(X)$	$X \cdot \texttt{c}$	construction component
			$\texttt{splicedc}[m; n; \grave{\kappa}]$	$\texttt{splicedc}[m; n; \grave{\kappa}]$	spliced construction

Syntax – Proto-Expressions and Proto-Rules

Sort			Operational Form	Stylized Form	Description
PrExp	$\grave{e}$	$::=$	x	x	variable
			$\texttt{prasc}\{\grave{\tau}\}(\grave{e})$	$\grave{e} : \grave{\tau}$	ascription
			$\texttt{prletval}(\grave{e}; x.\grave{e})$	$\texttt{let val}\ x = \grave{e}\ \texttt{in}\ \grave{e}$	value binding
			$\texttt{prlam}\{\grave{\tau}\}(x.\grave{e})$	$\lambda x{:}\grave{\tau}.\grave{e}$	abstraction
			$\texttt{prap}(\grave{e}; \grave{e})$	$\grave{e}(\grave{e})$	application
			$\texttt{prclam}\{\grave{\kappa}\}(u.\grave{e})$	$\Lambda u{::}\grave{\kappa}.\grave{e}$	construction abstraction
			$\texttt{prcap}\{\grave{c}\}(\grave{e})$	$\grave{e}[\grave{c}]$	construction application
			$\texttt{prfold}(\grave{e})$	$\texttt{fold}(\grave{e})$	fold
			$\texttt{prunfold}(\grave{e})$	$\texttt{unfold}(\grave{e})$	unfold
			$\texttt{prtpl}\{L\}(\{i \hookrightarrow \grave{e}_i\}_{i \in L})$	$\langle \{i \hookrightarrow \grave{e}_i\}_{i \in L} \rangle$	labeled tuple
			$\texttt{prprj}[\ell](\grave{e})$	$\grave{e} \cdot \ell$	projection
			$\texttt{prinj}[\ell](\grave{e})$	$\texttt{inj}[\ell](\grave{e})$	injection
			$\texttt{prmatch}[n](\grave{e}; \{\grave{r}_i\}_{1 \le i \le n})$	$\texttt{match}\ \grave{e}\ \{\grave{r}_i\}_{1 \le i \le n}$	match
			$\texttt{prval}(X)$	$X \cdot \texttt{v}$	value component
			$\texttt{splicede}[m; n; \grave{\tau}]$	$\texttt{splicede}[m; n; \grave{\tau}]$	spliced expression
PrRule	$\grave{r}$	$::=$	$\texttt{prrule}(p.\grave{e})$	$p \Rightarrow \grave{e}$	rule

Syntax – Proto-Patterns

PrPat	$\grave{p}$	$::=$	$\texttt{prwildp}$	$\_$	wildcard pattern
			$\texttt{prfoldp}(p)$	$\texttt{fold}(p)$	fold pattern
			$\texttt{prtplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i \in L})$	$\langle \{i \hookrightarrow \grave{p}_i\}_{i \in L} \rangle$	labeled tuple pattern
			$\texttt{prinjp}[\ell](\grave{p})$	$\texttt{inj}[\ell](\grave{p})$	injection pattern
			$\texttt{splicedp}[m; n; \grave{\tau}]$	$\texttt{splicedp}[m; n; \grave{\tau}]$	spliced pattern

Common Proto-Expansion Terms

Each expanded term, with a few exceptions noted below, maps onto a proto-expansion term. We refer to these as the *common proto-expansion terms*. In particular:

- Each kind, κ, maps onto a proto-kind, $\mathcal{P}(\kappa)$, as follows:

$$\begin{aligned}
\mathcal{P}(\texttt{darr}(\kappa_1; u.\kappa_2)) &= \texttt{prdarr}(\mathcal{P}(\kappa_1); u.\mathcal{P}(\kappa_2)) \\
\mathcal{P}(\texttt{unit}) &= \texttt{prunit} \\
\mathcal{P}(\texttt{dprod}(\kappa_1; u.\kappa_2)) &= \texttt{prdprod}(\mathcal{P}(\kappa_1); u.\mathcal{P}(\kappa_2)) \\
\mathcal{P}(\texttt{Type}) &= \texttt{prType} \\
\mathcal{P}(\texttt{S}(\tau)) &= \texttt{prS}(\mathcal{P}(\tau))
\end{aligned}$$

- Each construction, c, maps onto a proto-construction, $\mathcal{P}(c)$, as follows:

$$\begin{aligned}
\mathcal{P}(u) &= u \\
\mathcal{P}(\texttt{abs}(u.c)) &= \texttt{prabs}(u.\mathcal{P}(c)) \\
\mathcal{P}(\texttt{app}(c_1;c_2)) &= \texttt{prapp}(\mathcal{P}(c_1);\mathcal{P}(c_2)) \\
\mathcal{P}(\texttt{triv}) &= \texttt{prtriv} \\
\mathcal{P}(\texttt{pair}(c_1;c_2)) &= \texttt{prpair}(\mathcal{P}(c_1);\mathcal{P}(c_2)) \\
\mathcal{P}(\texttt{prl}(c)) &= \texttt{prprl}(\mathcal{P}(c)) \\
\mathcal{P}(\texttt{prr}(c)) &= \texttt{prprr}(\mathcal{P}(c)) \\
\mathcal{P}(\texttt{parr}(\tau_1;\tau_2)) &= \texttt{prparr}(\mathcal{P}(\tau_1);\mathcal{P}(\tau_2)) \\
\mathcal{P}(\texttt{all}(t.\tau)) &= \texttt{prall}(t.\mathcal{P}(\tau)) \\
\mathcal{P}(\texttt{rec}(t.\tau)) &= \texttt{prrec}(t.\mathcal{P}(\tau)) \\
\mathcal{P}(\texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})) &= \texttt{prprod}[L](\{i \hookrightarrow \mathcal{P}(\tau_i)\}_{i\in L}) \\
\mathcal{P}(\texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i\in L})) &= \texttt{prsum}[L](\{i \hookrightarrow \mathcal{P}(\tau_i)\}_{i\in L}) \\
\mathcal{P}(\texttt{con}(X)) &= \texttt{prcon}(X)
\end{aligned}$$

- Each expanded expression, e, except for the value projection of a module expression that is not of module variable form, maps onto a proto-expression, $\mathcal{P}(e)$, as follows:

$$\begin{aligned}
\mathcal{P}(x) &= x \\
\mathcal{P}(\texttt{lam}\{\tau\}(x.e)) &= \texttt{prlam}\{\mathcal{P}(\tau)\}(x.\mathcal{P}(e)) \\
\mathcal{P}(\texttt{ap}(e_1;e_2)) &= \texttt{prap}(\mathcal{P}(e_1);\mathcal{P}(e_2)) \\
\mathcal{P}(\texttt{clam}\{\kappa\}(u.e)) &= \texttt{prclam}\{\mathcal{P}(\kappa)\}(u.\mathcal{P}(e)) \\
\mathcal{P}(\texttt{cap}\{c\}(e)) &= \texttt{prcap}\{\mathcal{P}(c)\}(\mathcal{P}(e)) \\
\mathcal{P}(\texttt{fold}(e)) &= \texttt{prfold}(\mathcal{P}(e)) \\
\mathcal{P}(\texttt{unfold}(e)) &= \texttt{prunfold}(\mathcal{P}(e)) \\
\mathcal{P}(\texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L})) &= \texttt{prtpl}\{L\}(\{i \hookrightarrow \mathcal{P}(e_i)\}_{i\in L}) \\
\mathcal{P}(\texttt{inj}[\ell](e)) &= \texttt{prinj}[\ell](\mathcal{P}(e)) \\
\mathcal{P}(\texttt{match}[n](e;\{r_i\}_{1\le i\le n})) &= \texttt{prmatch}[n](\mathcal{P}(e);\{\mathcal{P}(r_i)\}_{1\le i\le n}) \\
\mathcal{P}(\texttt{val}(X)) &= \texttt{prval}(X)
\end{aligned}$$

- Each expanded rule, r, maps onto the proto-rule, $\mathcal{P}(r)$, as follows:

$$\mathcal{P}(\texttt{rule}(p.e)) = \texttt{prrule}(p.\mathcal{P}(e))$$

  Notice that proto-rules bind expanded patterns, not proto-patterns. This is because proto-rules appear in proto-expressions, which are generated by peTSMs. It would not be sensible for an peTSM to splice a pattern out of a literal body.

- Each expanded pattern, p, except for the variable patterns, maps onto a proto-pattern, $\mathcal{P}(p)$, as follows:

$$\begin{aligned}
\mathcal{P}(\texttt{wildp}) &= \texttt{prwildp} \\
\mathcal{P}(\texttt{foldp}(p)) &= \texttt{prfoldp}(\mathcal{P}(p)) \\
\mathcal{P}(\texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i\in L})) &= \texttt{prtplp}[L](\{i \hookrightarrow \mathcal{P}(p_i)\}_{i\in L}) \\
\mathcal{P}(\texttt{injp}[\ell](p)) &= \texttt{prinjp}[\ell](\mathcal{P}(p))
\end{aligned}$$

Parameterized Proto-Expression Encoding and Decoding

The type abbreviated `PPrExpr` classifies encodings of *parameterized proto-expressions*. The mapping from parameterized proto-expressions to values of type `PPrExpr` is defined by the *parameterized proto-expression encoding judgement*, $\dot{e} \downarrow_{\mathsf{PPrExpr}} e$. An inverse mapping is defined by the *parameterized proto-expression decoding judgement*, $e \uparrow_{\mathsf{PPrExpr}} \dot{e}$.

Judgement Form	**Description**
$\dot{e} \downarrow_{\mathsf{PPrExpr}} e$	$\dot{e}$ has encoding e
$e \uparrow_{\mathsf{PPrExpr}} \dot{e}$	e has decoding $\dot{e}$

Rather than picking a particular definition of `PPrExpr` and defining the judgements above inductively against it, we only state the following condition, which establishes an isomorphism between values of type `PPrExpr` and parameterized proto-expressions.

Condition C.21 (Parameterized Proto-Expression Isomorphism)**.**

1. *For every $\dot{e}$, we have $\dot{e} \downarrow_{\mathsf{PPrExpr}} e_{proto}$ for some e_{proto} such that $\vdash e_{proto} : \texttt{PPrExpr}$ and e_{proto} val.*
2. *If $\vdash e_{proto} : \texttt{PPrExpr}$ and e_{proto} val then $e_{proto} \uparrow_{\mathsf{PPrExpr}} \dot{e}$ for some $\dot{e}$.*
3. *If $\dot{e} \downarrow_{\mathsf{PPrExpr}} e_{proto}$ then $e_{proto} \uparrow_{\mathsf{PPrExpr}} \dot{e}$.*
4. *If $\vdash e_{proto} : \texttt{PPrExpr}$ and e_{proto} val and $e_{proto} \uparrow_{\mathsf{PPrExpr}} \dot{e}$ then $\dot{e} \downarrow_{\mathsf{PPrExpr}} e_{proto}$.*
5. *If $\dot{e} \downarrow_{\mathsf{PPrExpr}} e_{proto}$ and $\dot{e} \downarrow_{\mathsf{PPrExpr}} e'_{proto}$ then $e_{proto} = e'_{proto}$.*
6. *If $\vdash e_{proto} : \texttt{PPrExpr}$ and e_{proto} val and $e_{proto} \uparrow_{\mathsf{PPrExpr}} \dot{e}$ and $e_{proto} \uparrow_{\mathsf{PPrExpr}} \dot{e}'$ then $\dot{e} = \dot{e}'$.*

Parameterized Proto-Pattern Encoding and Decoding

The type abbreviated `PPrPat` classifies encodings of *parameterized proto-patterns*. The mapping from parameterized proto-patterns to values of type `PPrPat` is defined by the *parameterized proto-pattern encoding judgement*, $\dot{p} \downarrow_{\mathsf{PPrPat}} p$. An inverse mapping is defined by the *parameterized proto-expression decoding judgement*, $p \uparrow_{\mathsf{PPrPat}} \dot{p}$.

Judgement Form	**Description**
$\dot{p} \downarrow_{\mathsf{PPrPat}} p$	$\dot{p}$ has encoding p
$p \uparrow_{\mathsf{PPrPat}} \dot{p}$	p has decoding $\dot{p}$

Again, rather than picking a particular definition of `PPrPat` and defining the judgements above inductively against it, we only state the following condition, which establishes an isomorphism between values of type `PPrPat` and parameterized proto-patterns.

Condition C.22 (Parameterized Proto-Pattern Isomorphism)**.**

1. *For every $\dot{p}$, we have $\dot{p} \downarrow_{\mathsf{PPrPat}} e_{proto}$ for some e_{proto} such that $\vdash e_{proto} : \texttt{PPrPat}$ and e_{proto} val.*
2. *If $\vdash e_{proto} : \texttt{PPrPat}$ and e_{proto} val then $e_{proto} \uparrow_{\mathsf{PPrPat}} \dot{p}$ for some $\dot{p}$.*
3. *If $\dot{p} \downarrow_{\mathsf{PPrPat}} e_{proto}$ then $e_{proto} \uparrow_{\mathsf{PPrPat}} \dot{p}$.*
4. *If $\vdash e_{proto} : \texttt{PPrPat}$ and e_{proto} val and $e_{proto} \uparrow_{\mathsf{PPrPat}} \dot{p}$ then $\dot{p} \downarrow_{\mathsf{PPrPat}} e_{proto}$.*
5. *If $\dot{p} \downarrow_{\mathsf{PPrPat}} e_{proto}$ and $\dot{p} \downarrow_{\mathsf{PPrPat}} e'_{proto}$ then $e_{proto} = e'_{proto}$.*
6. *If $\vdash e_{proto} : \texttt{PPrPat}$ and e_{proto} val and $e_{proto} \uparrow_{\mathsf{PPrPat}} \dot{p}$ and $e_{proto} \uparrow_{\mathsf{PPrPat}} \dot{p}'$ then $\dot{p} = \dot{p}'$.*

Splice Summaries

The *splice summary* of a proto-expression, $\mathsf{summary}(\grave{e})$, or proto-pattern, $\mathsf{summary}(\grave{p})$, is the finite set of references to spliced kinds, constructions, expressions and patterns that it mentions.

Segmentations

A *segment set*, ψ, is a finite set of pairs of natural numbers indicating the locations of spliced terms. The *segmentation* of a proto-expression, $\mathsf{seg}(\grave{e})$, or proto-pattern, $\mathsf{seg}(\grave{p})$, is the segment set implied by the splice summary.

The predicate ψ $\mathsf{segments}$ b checks that each segment in ψ, has non-negative length and is within bounds of b, and that the segments in ψ do not overlap.

C.3.2 Deparameterization

$\boxed{\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \dot{e} \looparrowright_{\epsilon} \grave{e} \mathrel{?} \rho \dashv \omega : \Omega_{\text{params}}}$ When applying peTSM ϵ, $\dot{e}$ has deparameterization $\grave{e}$ leaving ρ with parameter substitution ω

$$\frac{\Omega_{\text{app}} \vdash \rho \;\mathsf{tsmty}}{\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi, a \hookrightarrow \mathtt{petsm}(\rho; e_{\text{parse}})} \mathtt{prexp}(\grave{e}) \looparrowright_{\mathtt{defref}[a]} \grave{e} \mathrel{?} \rho \dashv \emptyset : \emptyset} \tag{C.37a}$$

$$\frac{\begin{array}{c}\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \dot{e} \looparrowright_{\epsilon} \grave{e} \mathrel{?} \mathtt{allmods}\{\sigma\}(X.\rho) \dashv \omega : \Omega \\ \Omega_{\text{app}} \vdash X' : \sigma \qquad X \notin \mathrm{dom}(\Omega_{\text{app}})\end{array}}{\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{prbindmod}(X.\dot{e}) \looparrowright_{\mathtt{apmod}\{X'\}(\epsilon)} \grave{e} \mathrel{?} \rho \dashv (\omega, X'/X) : (\Omega, X : \sigma)} \tag{C.37b}$$

$\boxed{\Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi} \dot{p} \looparrowright_{\epsilon} \grave{p} \mathrel{?} \rho \dashv \omega : \Omega_{\text{params}}}$ When applying ppTSM ϵ, $\dot{p}$ has deparameterization $\grave{p}$ leaving ρ with parameter substitution ω

$$\frac{\Omega_{\text{app}} \vdash \rho \;\mathsf{tsmty}}{\Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi, a \hookrightarrow \mathtt{pptsm}(\rho; e_{\text{parse}})} \mathtt{prpat}(\grave{p}) \looparrowright_{\mathtt{defref}[a]} \grave{p} \mathrel{?} \rho \dashv \emptyset : \emptyset} \tag{C.38a}$$

$$\frac{\begin{array}{c}\Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi} \dot{p} \looparrowright_{\epsilon} \grave{p} \mathrel{?} \mathtt{allmods}\{\sigma\}(X.\rho) \dashv \omega : \Omega \\ \Omega_{\text{app}} \vdash X' : \sigma \qquad X \notin \mathrm{dom}(\Omega_{\text{app}})\end{array}}{\Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi} \mathtt{prbindmod}(X.\dot{p}) \looparrowright_{\mathtt{apmod}\{X'\}(\epsilon)} \grave{p} \mathrel{?} \rho \dashv (\omega, X'/X) : (\Omega, X : \sigma)} \tag{C.38b}$$

C.3.3 Proto-Expansion Validation

Splicing Scenes

Expression splicing scenes, $\mathbb{E}$, are of the form $\omega : \Omega_{\text{params}}; \hat{\Omega}; \hat{\Psi}; \hat{\Phi}; b$, *construction splicing scenes*, $\mathbb{C}$, are of the form $\omega : \Omega_{\text{params}}; \hat{\Omega}; b$, and *pattern splicing scenes*, $\mathbb{P}$, are of the form $\omega : \Omega_{\text{params}}; \hat{\Omega}; \hat{\Phi}; b$. We write $\mathsf{cs}(\mathbb{E})$ for the construction splicing scene constructed by dropping the TSM contexts from $\mathbb{E}$:

$$\mathsf{cs}(\omega : \Omega_{\text{params}}; \hat{\Omega}; \hat{\Psi}; \hat{\Phi}; b) = \omega : \Omega_{\text{params}}; \hat{\Omega}; b$$

Proto-Kind and Proto-Construction Validation

$\boxed{\Omega \vdash^{\mathbb{C}} \grave{\kappa} \rightsquigarrow \kappa \ \mathsf{kind}}$ $\grave{\kappa}$ has well-formed expansion κ

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{\kappa}_1 \rightsquigarrow \kappa_1 \ \mathsf{kind} \qquad \Omega, u :: \kappa_1 \vdash^{\mathbb{C}} \grave{\kappa}_2 \rightsquigarrow \kappa_2 \ \mathsf{kind}}{\Omega \vdash^{\mathbb{C}} \mathtt{prdarr}(\grave{\kappa}_1; u.\grave{\kappa}_2) \rightsquigarrow \mathtt{darr}(\kappa_1; u.\kappa_2) \ \mathsf{kind}} \tag{C.39a}$$

$$\frac{}{\Omega \vdash^{\mathbb{C}} \mathtt{prunit} \rightsquigarrow \mathtt{unit} \ \mathsf{kind}} \tag{C.39b}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{\kappa}_1 \rightsquigarrow \kappa_1 \ \mathsf{kind} \qquad \Omega, u :: \kappa_1 \vdash^{\mathbb{C}} \grave{\kappa}_2 \rightsquigarrow \kappa_2 \ \mathsf{kind}}{\Omega \vdash^{\mathbb{C}} \mathtt{prdprod}(\grave{\kappa}_1; u.\grave{\kappa}_2) \rightsquigarrow \mathtt{dprod}(\kappa_1; u.\kappa_2) \ \mathsf{kind}} \tag{C.39c}$$

$$\frac{}{\Omega \vdash^{\mathbb{C}} \mathtt{prType} \rightsquigarrow \mathtt{Type} \ \mathsf{kind}} \tag{C.39d}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{\tau} \rightsquigarrow \tau :: \mathtt{Type}}{\Omega \vdash^{\mathbb{C}} \mathtt{prS}(\grave{\tau}) \rightsquigarrow \mathtt{S}(\tau) \ \mathsf{kind}} \tag{C.39e}$$

$$\frac{\begin{array}{c}\mathsf{parseUKind}(\mathsf{subseq}(b; m; n)) = \hat{\kappa} \qquad \hat{\Omega} \vdash \hat{\kappa} \rightsquigarrow \kappa \ \mathsf{kind} \\ \hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\mathrm{app}} \rangle \qquad \mathrm{dom}(\Omega) \cap \mathrm{dom}(\Omega_{\mathrm{app}}) = \emptyset\end{array}}{\Omega \vdash^{\omega:\Omega_{\mathrm{params}}; \hat{\Omega}; b} \mathtt{splicedk}[m; n] \rightsquigarrow \kappa \ \mathsf{kind}} \tag{C.39f}$$

$\boxed{\Omega \vdash^{\mathbb{C}} \grave{c} \rightsquigarrow c :: \kappa}$ $\grave{c}$ has expansion c of kind κ

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{c} \rightsquigarrow c :: \kappa_1 \qquad \Omega \vdash \kappa_1 <:: \kappa_2}{\Omega \vdash^{\mathbb{C}} \grave{c} \rightsquigarrow c :: \kappa_2} \tag{C.40a}$$

$$\frac{}{\Omega, u :: \kappa \vdash^{\mathbb{C}} u \rightsquigarrow u :: \kappa} \tag{C.40b}$$

$$\frac{\Omega, u :: \kappa_1 \vdash^{\mathbb{C}} \grave{c}_2 \rightsquigarrow c_2 :: \kappa_2}{\Omega \vdash^{\mathbb{C}} \mathtt{prabs}(u.\grave{c}_2) \rightsquigarrow \mathtt{abs}(u.c_2) :: \mathtt{darr}(\kappa_1; u.\kappa_2)} \tag{C.40c}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{c}_1 \rightsquigarrow c_1 :: \mathtt{darr}(\kappa_2; u.\kappa) \qquad \Omega \vdash^{\mathbb{C}} \grave{c}_2 \rightsquigarrow c_2 :: \kappa_2}{\Omega \vdash^{\mathbb{C}} \mathtt{prapp}(\grave{c}_1; \grave{c}_2) \rightsquigarrow \mathtt{app}(c_1; c_2) :: [c_1/u]\kappa} \tag{C.40d}$$

$$\frac{}{\Omega \vdash^{\mathbb{C}} \mathtt{prtriv} \rightsquigarrow \mathtt{triv} :: \mathtt{unit}} \tag{C.40e}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{c}_1 \rightsquigarrow c_1 :: \kappa_1 \qquad \Omega \vdash^{\mathbb{C}} \grave{c}_2 \rightsquigarrow c_2 :: [c_1/u]\kappa_2}{\Omega \vdash^{\mathbb{C}} \mathtt{prpair}(\grave{c}_1; \grave{c}_2) \rightsquigarrow \mathtt{pair}(c_1; c_2) :: \mathtt{dprod}(\kappa_1; u.\kappa_2)} \tag{C.40f}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{c} \rightsquigarrow c :: \mathtt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash^{\mathbb{C}} \mathtt{prprl}(\grave{c}) \rightsquigarrow \mathtt{prl}(c) :: \kappa_1} \tag{C.40g}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{c} \leadsto c :: \mathtt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash^{\mathbb{C}} \mathtt{prprr}(\grave{c}) \leadsto \mathtt{prr}(c) :: [\mathtt{prl}(c)/u]\kappa_2} \tag{C.40h}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{\tau}_1 \leadsto \tau_1 :: \mathsf{Type} \qquad \Omega \vdash^{\mathbb{C}} \grave{\tau}_2 \leadsto \tau_2 :: \mathsf{Type}}{\Omega \vdash^{\mathbb{C}} \mathtt{prparr}(\grave{\tau}_1; \grave{\tau}_2) \leadsto \mathtt{parr}(\tau_1; \tau_2) :: \mathsf{Type}} \tag{C.40i}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{\kappa} \leadsto \kappa \; \mathsf{kind} \qquad \Omega, u :: \kappa \vdash^{\mathbb{C}} \grave{\tau} \leadsto \tau :: \mathsf{Type}}{\Omega \vdash^{\mathbb{C}} \mathtt{prall}\{\grave{\kappa}\}(u.\grave{\tau}) \leadsto \mathtt{all}\{\kappa\}(u.\tau) :: \mathsf{Type}} \tag{C.40j}$$

$$\frac{\Omega, t :: \mathsf{Type} \vdash^{\mathbb{C}} \grave{\tau} \leadsto \tau :: \mathsf{Type}}{\Omega \vdash^{\mathbb{C}} \mathtt{prrec}(t.\grave{\tau}) \leadsto \mathtt{rec}(t.\tau) :: \mathsf{Type}} \tag{C.40k}$$

$$\frac{\{\Omega \vdash^{\mathbb{C}} \grave{\tau}_i \leadsto \tau_i :: \mathsf{Type}\}_{1 \le i \le n}}{\Omega \vdash^{\mathbb{C}} \mathtt{prprod}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}) \leadsto \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) :: \mathsf{Type}} \tag{C.40l}$$

$$\frac{\{\Omega \vdash^{\mathbb{C}} \grave{\tau}_i \leadsto \tau_i :: \mathsf{Type}\}_{1 \le i \le n}}{\Omega \vdash^{\mathbb{C}} \mathtt{prsum}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}) \leadsto \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) :: \mathsf{Type}} \tag{C.40m}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{c} \leadsto c :: \mathsf{Type}}{\Omega \vdash^{\mathbb{C}} \grave{c} \leadsto c :: \mathtt{S}(c)} \tag{C.40n}$$

$$\frac{}{\Omega, X : \mathtt{sig}\{\kappa\}(u.\tau) \vdash^{\mathbb{C}} \mathtt{prcon}(X) \leadsto \mathtt{con}(X) :: \kappa} \tag{C.40o}$$

$$\frac{\begin{array}{c}\mathbb{C} = \omega : \Omega_{\text{params}}; \hat{\Omega}; b \qquad \Omega_{\text{params}} \vdash^{\mathbb{C}} \grave{\kappa} \leadsto \kappa \; \mathsf{kind} \\ \mathsf{parseUCon}(\mathsf{subseq}(b; m; n)) = \hat{c} \qquad \hat{\Omega} \vdash \hat{c} \leadsto c :: [\omega]\kappa \\ \hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\text{app}} \rangle \qquad \mathrm{dom}(\Omega) \cap \mathrm{dom}(\Omega_{\text{app}}) = \emptyset\end{array}}{\Omega \vdash^{\mathbb{C}} \mathtt{splicedc}[m; n; \grave{\kappa}] \leadsto c :: \kappa} \tag{C.40p}$$

Proto-Expression and Proto-Rule Validation

$\boxed{\Omega \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau}$ $\grave{e}$ has expansion e of type τ

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau \qquad \Omega \vdash \tau <: \tau'}{\Omega \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau'} \tag{C.41a}$$

$$\frac{}{\Omega, x : \tau \vdash^{\mathbb{E}} x \leadsto x : \tau} \tag{C.41b}$$

$$\frac{\Omega \vdash^{\mathsf{cs}(\mathbb{E})} \grave{\tau} \leadsto \tau :: \mathsf{Type} \qquad \Omega \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau}{\Omega \vdash^{\mathbb{E}} \mathtt{prasc}\{\grave{\tau}\}(\grave{e}) \leadsto e : \tau} \tag{C.41c}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e}_1 \leadsto e_1 : \tau_1 \qquad \Omega, x : \tau_1 \vdash^{\grave{e}_2} e_2 \leadsto \tau_2 :}{\Omega \vdash^{\mathbb{E}} \mathtt{prletval}(\grave{e}_1; x.\grave{e}_2) \leadsto \mathtt{ap}(\mathtt{lam}\{\tau_1\}(x.e_2); e_1) : \tau_2} \tag{C.41d}$$

$$\frac{\Omega \vdash^{\mathsf{cs}(\mathbb{E})} \grave{\tau}_1 \leadsto \tau_1 :: \mathtt{Type} \qquad \Omega, x : \tau_1 \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau_2}{\Omega \vdash^{\mathbb{E}} \mathtt{prlam}\{\grave{\tau}_1\}(x.\grave{e}) \leadsto \mathtt{lam}\{\tau_1\}(x.e) : \mathtt{parr}(\tau_1; \tau_2)} \tag{C.41e}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e}_1 \leadsto e_1 : \mathtt{parr}(\tau_2; \tau) \qquad \Omega \vdash^{\mathbb{E}} \grave{e}_2 \leadsto e_2 : \tau_2}{\Omega \vdash^{\mathbb{E}} \mathtt{prap}(\grave{e}_1; \grave{e}_2) \leadsto \mathtt{ap}(e_1; e_2) : \tau} \tag{C.41f}$$

$$\frac{\Omega \vdash^{\mathsf{cs}(\mathbb{E})} \grave{\kappa} \leadsto \kappa \ \mathsf{kind} \qquad \Omega, u :: \kappa \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prclam}\{\grave{\kappa}\}(u.\grave{e}) \leadsto \mathtt{clam}\{\kappa\}(u.e) \Rightarrow \mathtt{all}\{\kappa\}(u.\tau)} \tag{C.41g}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \leadsto e : \mathtt{all}\{\kappa\}(u.\tau) \qquad \Omega \vdash^{\mathsf{cs}(\mathbb{E})} \grave{c} \leadsto c :: \kappa}{\Omega \vdash^{\mathbb{E}} \mathtt{prcap}\{\grave{c}\}(\grave{e}) \leadsto \mathtt{cap}\{c\}(e) : [c/u]\tau} \tag{C.41h}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \leadsto e : [\mathtt{rec}(t.\tau)/t]\tau}{\Omega \vdash^{\mathbb{E}} \mathtt{prfold}(\grave{e}) \leadsto \mathtt{fold}(e) : \mathtt{rec}(t.\tau)} \tag{C.41i}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \leadsto e : \mathtt{rec}(t.\tau)}{\Omega \vdash^{\mathbb{E}} \mathtt{prunfold}(\grave{e}) \leadsto \mathtt{unfold}(e) : [\mathtt{rec}(t.\tau)/t]\tau} \tag{C.41j}$$

$$\frac{\begin{array}{c}\tau = \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \\ \{\Omega \vdash^{\mathbb{E}} \grave{e}_i \leadsto e_i : \tau_i\}_{i \in L}\end{array}}{\Omega \vdash^{\mathbb{E}} \mathtt{prtpl}\{L\}(\{i \hookrightarrow \grave{e}_i\}_{i \in L}) \leadsto \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) : \tau} \tag{C.41k}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \leadsto e : \mathtt{prod}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)}{\Omega \vdash^{\mathbb{E}} \mathtt{prprj}[\ell](\grave{e}) \leadsto \mathtt{prj}[\ell](e) : \tau} \tag{C.41l}$$

$$\frac{\begin{array}{c}\mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau') \\ \Omega \vdash^{\mathbb{E}} \grave{e}' \leadsto e' : \tau'\end{array}}{\Omega \vdash^{\mathbb{E}} \mathtt{prinj}[\ell](\grave{e}') \leadsto \mathtt{inj}[\ell](e') : \tau} \tag{C.41m}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau \qquad \{\Omega \vdash^{\mathbb{E}} \grave{r}_i \leadsto r_i : \tau \Mapsto \tau'\}_{1 \le i \le n}}{\Omega \vdash^{\mathbb{E}} \mathtt{prmatch}[n](\grave{e}; \{\grave{r}_i\}_{1 \le i \le n}) \leadsto \mathtt{match}[n](e; \{r_i\}_{1 \le i \le n}) : \tau'} \tag{C.41n}$$

$$\frac{}{\Omega, X : \mathtt{sig}\{\kappa\}(u.\tau) \vdash^{\mathbb{E}} \mathtt{prval}(X) \leadsto \mathtt{val}(X) : [\mathtt{con}(X)/u]\tau} \tag{C.41o}$$

$$\frac{\begin{array}{c}\mathbb{E} = \omega : \Omega_{\text{params}}; \hat{\Omega}; \hat{\Psi}; \hat{\Phi}; b \qquad \Omega_{\text{params}} \vdash^{\mathsf{cs}(\mathbb{E})} \grave{\tau} \leadsto \tau :: \mathtt{Type} \\ \mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e} \qquad \hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \leadsto e : [\omega]\tau \\ \hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\text{app}} \rangle \qquad \mathrm{dom}(\Omega) \cap \mathrm{dom}(\Omega_{\text{app}}) = \emptyset\end{array}}{\Omega \vdash^{\mathbb{E}} \mathtt{splicede}[m; n; \grave{\tau}] \leadsto e : \tau} \tag{C.41p}$$

$\boxed{\Omega \vdash^{\mathbb{E}} \grave{r} \leadsto r : \tau \Mapsto \tau'}$ $\grave{r}$ has expansion r taking values of type τ to values of type τ'

$$\frac{\Omega \vdash p : \tau \dashv\!\!| \ \Omega' \qquad \Omega \cup \Omega' \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau'}{\Omega \vdash^{\mathbb{E}} \mathtt{prrule}(p.\grave{e}) \leadsto \mathtt{rule}(p.e) : \tau \Mapsto \tau'} \tag{C.42}$$

Proto-Pattern Validation

$\boxed{\grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\mathbb{P}} \hat{\Omega}}$ $\grave{p}$ has expansion p matching against τ generating hypotheses $\hat{\Omega}$

$$\frac{}{\texttt{prwildp} \rightsquigarrow \texttt{wildp} : \tau \dashv\!\mid^{\mathbb{P}} \langle\emptyset;\emptyset;\emptyset;\emptyset\rangle} \tag{C.43a}$$

$$\frac{\grave{p} \rightsquigarrow p : [\texttt{rec}(t.\tau)/t]\tau \dashv\!\mid^{\mathbb{P}} \hat{\Omega}}{\texttt{prfoldp}(\grave{p}) \rightsquigarrow \texttt{foldp}(p) : \texttt{rec}(t.\tau) \dashv\!\mid^{\mathbb{P}} \hat{\Omega}} \tag{C.43b}$$

$$\frac{\grave{p} = \texttt{prtplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i\in L}) \qquad p = \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i\in L}) \qquad \{\grave{p}_i \rightsquigarrow p_i : \tau_i \dashv\!\mid^{\mathbb{P}} \hat{\Gamma}_i\}_{i\in L}}{\grave{p} \rightsquigarrow p : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L}) \dashv\!\mid^{\mathbb{P}} \uplus_{i\in L}\hat{\Omega}_i} \tag{C.43c}$$

$$\frac{\grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\mathbb{P}} \hat{\Omega}}{\texttt{prinjp}[\ell](\grave{p}) \rightsquigarrow \texttt{injp}[\ell](p) : \texttt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i\in L}; \ell \hookrightarrow \tau) \dashv\!\mid^{\mathbb{P}} \hat{\Omega}} \tag{C.43d}$$

$$\frac{\Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}};\hat{\Omega};b} \grave{\tau} \rightsquigarrow \tau :: \texttt{Type} \qquad \mathsf{parseUPat}(\mathsf{subseq}(b;m;n)) = \hat{p} \qquad \hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : [\omega]\tau \dashv\!\mid \hat{\Omega}'}{\texttt{splicedp}[m;n;\grave{\tau}] \rightsquigarrow p : \tau \dashv\!\mid^{\omega:\Omega_{\text{params}};\hat{\Omega};\hat{\Phi};b} \hat{\Omega}'} \tag{C.43e}$$

C.4 Metatheory

C.4.1 TSM Expressions

Lemma C.23 (peTSM Regularity). *If* $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} \rho$ *then* $\Omega \vdash \rho$ tsmty.

Proof. By rule induction over Rules (C.26).

Case (C.26a).

(1) $\Omega \vdash \rho$ tsmty	by assumption

Case (C.26b).

(1) $\epsilon = \texttt{absmod}\{\sigma\}(X.\epsilon')$	by assumption
(2) $\rho = \texttt{allmods}\{\sigma\}(X.\rho')$	by assumption
(3) $\Omega, X : \sigma \vdash^{\mathsf{Exp}}_{\Psi} \epsilon' \mathbin{@} \rho'$	by assumption
(4) $\Omega \vdash \sigma$ sig	by assumption
(5) $\Omega, X : \sigma \vdash \rho'$ tsmty	by IH on (3)
(6) $\Omega \vdash \texttt{allmods}\{\sigma\}(X.\rho')$ tsmty	by Rule (C.25b) on (4) and (5)

Case (C.26c).

(1) $\epsilon = \texttt{apmod}\{X\}(\epsilon')$	by assumption
(2) $\rho = [X/X']\rho'$	by assumption
(3) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} \texttt{allmods}\{\sigma\}(X'.\rho')$	by assumption
(4) $\Omega \vdash X : \sigma$	by assumption
(5) $\Omega \vdash \texttt{allmods}\{\sigma\}(X'.\rho')$ tsmty	by IH on (3)

(6) $\Omega, X' : \sigma \vdash \rho'$ tsmty — by Inversion of Rule (C.25b) on (5)

(7) $\Omega \vdash [X'/X]\rho'$ tsmty — by Substitution Lemma C.3 on (4) and (6)

$\square$

Lemma C.24 (ppTSM Regularity). *If* $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mathbin{@} \rho$ *then* $\Omega \vdash \rho$ tsmty.
Proof. By rule induction over Rules (C.27). The proof is nearly identical to the proof of Lemma C.23, differing only in that ppTSM contexts and the corresponding judgements are mentioned. $\square$

Lemma C.25 (peTSM Unicity). *If* $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} \rho$ *and* $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} \rho'$ *then* $\rho = \rho'$.
Proof. By rule induction over Rules (C.26). The rules are syntax-directed, so the proof is by straightforward observations of syntactic contradictions. $\square$

Lemma C.26 (ppTSM Unicity). *If* $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mathbin{@} \rho$ *and* $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mathbin{@} \rho'$ *then* $\rho = \rho'$.
Proof. By rule induction over Rules (C.27). The rules are syntax-directed, so the proof is by straightforward observations of syntactic contradictions. $\square$

Theorem C.27 (peTSM Preservation). *If* $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} \rho$ *and* $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto \epsilon'$ *then* $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon' \mathbin{@} \rho$.
Proof. By rule induction over Rules (C.29).

Case (C.29a). By rule induction over Rules (C.26). There is only one rule that applies.

Case (C.26c).

(1) $\epsilon = \mathtt{apmod}\{X\}(\epsilon'')$	by assumption
(2) $\epsilon' = \mathtt{apmod}\{X\}(\epsilon''')$	by assumption
(3) $\rho = [X/X']\rho'$	by assumption
(4) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon'' \mapsto \epsilon'''$	by assumption
(5) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon'' \mathbin{@} \mathtt{allmods}\{\sigma\}(X'.\rho')$	by assumption
(6) $\Omega \vdash X : \sigma$	by assumption
(7) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon''' \mathbin{@} \mathtt{allmods}\{\sigma\}(X'.\rho')$	by IH on (5) and (4)
(8) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{apmod}\{X\}(\epsilon''') \mathbin{@} [X/X']\rho$	by Rule (C.26c) on (7) and (6)

Case (C.29b). By rule induction over Rules (C.26). There is only one rule that applies.

Case (C.26c).

(1) $\epsilon = \mathtt{apmod}\{X\}(\mathtt{absmod}\{\sigma\}(X'.\epsilon''))$	by assumption
(2) $\epsilon' = [X/X']\epsilon''$	by assumption
(3) $\rho = [X/X']\rho'$	by assumption
(4) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{absmod}\{\sigma\}(X'.\epsilon'') \mathbin{@} \mathtt{allmods}\{\sigma\}(X'.\rho')$	by assumption
(5) $\Omega \vdash X : \sigma$	by assumption
(6) $\Omega, X' : \sigma \vdash^{\mathsf{Exp}}_{\Psi} \epsilon'' \mathbin{@} \rho'$	by Inversion Lemma for Rule (C.26b) on (4)

(7) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} [X/X']\epsilon'' @ [X/X']\rho'$ by Substitution Lemma C.3 on (6)

□

Corollary C.28 (peTSM Preservation (Multistep)). *If* $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon @ \rho$ *and* $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto^{*} \epsilon'$ *then* $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon' @ \rho$.

Proof. The multistep relation is the reflexive, transitive closure of the single step relation, so the proof follows by applying Theorem C.27 over each step. □

Corollary C.29 (peTSM Preservation (Evaluation)). *If* $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon @ \rho$ *and* $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \Downarrow \epsilon'$ *then* $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon' @ \rho$.

Proof. The evaluation relation is the multistep relation with an additional requirement, so the proof follows directly from Corollary C.28. □

Theorem C.30 (ppTSM Preservation). *If* $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon @ \rho$ *and* $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mapsto \epsilon'$ *then* $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon' @ \rho$.

Proof. The proof is nearly identical to the proof of Theorem C.27, differing only in that ppTSM contexts and the corresponding judgements are mentioned. □

Corollary C.31 (ppTSM Preservation (Multistep)). *If* $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon @ \rho$ *and* $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mapsto^{*} \epsilon'$ *then* $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon' @ \rho$.

Proof. The multistep relation is the reflexive, transitive closure of the single step relation, so the proof follows by applying Theorem C.30 over each step. □

Corollary C.32 (ppTSM Preservation (Evaluation)). *If* $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon @ \rho$ *and* $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \Downarrow \epsilon'$ *then* $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon' @ \rho$.

Proof. The evaluation relation is the multistep relation with an additional requirement, so the proof follows directly from Corollary C.31. □

Theorem C.33 (peTSM Progress). *If* $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon @ \rho$ *then either* $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto \epsilon'$ *for some* ϵ' *or* $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon$ normal.

Proof. By rule induction over Rules (C.26).

Case (C.26a).

(1) $\epsilon = \mathtt{defref}[a]$ by assumption

(2) $\Psi = \Psi', a \hookrightarrow \mathtt{petsm}(\rho; e_{\text{parse}})$ by assumption

(3) $\Omega \vdash^{\mathsf{Exp}}_{\Psi', a \hookrightarrow \mathtt{petsm}(\rho; e_{\text{parse}})} \mathtt{defref}[a]$ normal by Rule (C.35a)

Case (C.26b).

(1) $\epsilon = \mathtt{absmod}\{\sigma\}(X.\epsilon')$ by assumption

(2) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{absmod}\{\sigma\}(X.\epsilon')$ normal by Rule (C.35b)

Case (C.26c).

(1) $\epsilon = \mathtt{apmod}\{X\}(\epsilon')$ by assumption

(2) $\rho = [X/X']\rho'$ by assumption

(3) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon' @ \mathtt{allmods}\{\sigma\}(X'.\rho')$ by assumption

(4) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon' \mapsto \epsilon''$ for some ϵ'' or $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon'$ normal by IH on (3)

Proceed by cases on (4).

Case $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon' \mapsto \epsilon''$**.**

(5) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon' \mapsto \epsilon''$	by assumption
(6) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{apmod\{}X\mathtt{\}(}\epsilon'\mathtt{)} \mapsto \mathtt{apmod\{}X\mathtt{\}(}\epsilon''\mathtt{)}$	by Rule (C.29a) on (5)

Case $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon'$ normal**.** Proceed by rule induction over Rules (C.35).

Case (C.35a)**.**

(7) $\epsilon' = \mathtt{defref[}a\mathtt{]}$	by assumption
(8) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon'$ normal	by assumption
(9) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{apmod\{}X\mathtt{\}(}\epsilon'\mathtt{)}$ normal	by Rule (C.35c) on (8)

Case (C.35b)**.**

(10) $\epsilon' = \mathtt{absmod\{}\sigma\mathtt{\}(}X.\epsilon''\mathtt{)}$	by assumption
(11) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{apmod\{}X\mathtt{\}(absmod\{}\sigma\mathtt{\}(}X'.\epsilon''\mathtt{))} \mapsto [X/X']\epsilon''$	by Rule (C.29b)

Case (C.35c)**.**

(12) $\epsilon' = \mathtt{apmod\{}X''\mathtt{\}(}\epsilon''\mathtt{)}$	by assumption
(13) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon'$ normal	by assumption
(14) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{apmod\{}X\mathtt{\}(}\epsilon'\mathtt{)}$ normal	by Rule (C.35c) on (8)

□

Theorem C.34 (ppTSM Progress)**.** *If* $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mathbin{@} \rho$ *then either* $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mapsto \epsilon'$ *for some* ϵ' *or* $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon$ normal.

Proof. The proof is nearly identical to the proof of Theorem C.33, differing only in that ppTSM contexts and the corresponding judgements are mentioned. □

C.4.2 Typed Expansion

Kinds, Constructions and Signatures

Theorem C.35 (Kind and Construction Expansion)**.**

1. *If* $\langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle \vdash \hat{\kappa} \rightsquigarrow \kappa$ kind *then* $\Omega \vdash \kappa$ kind.
2. *If* $\langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle \vdash \hat{c} \rightsquigarrow c :: \kappa$ *then* $\Omega \vdash c :: \kappa$.

Proof. By mutual rule induction over Rules (C.17) and Rules (C.18). In each case, we apply the IH to each premise and then apply the corresponding kind formation rule from Rules (C.6) or kinding rule from Rules (C.9). □

Theorem C.36 (Signature Expansion)**.** *If* $\langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle \vdash \hat{\sigma} \rightsquigarrow \sigma$ sig *then* $\Omega \vdash \sigma$ sig.

Proof. By rule induction over Rule (C.15). Apply Theorem C.35 to each premise, then apply Rule (C.1). □

TSM Types and Expressions

Theorem C.37 (TSM Type Expansion)**.** *If* $\langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle \vdash \hat{\rho} \rightsquigarrow \rho$ tsmty *then* $\Omega \vdash \rho$ tsmty.

Proof. By rule induction over Rules (C.22).

Case (C.22a).

(1) $\hat{\rho} = \hat{\tau}$ by assumption

(2) $\rho = \texttt{type}(\tau)$ by assumption

(3) $\hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau :: \texttt{Type}$ by assumption

(4) $\Omega \vdash \tau :: \texttt{Type}$ by Theorem C.35 on (3)

(5) $\Omega \vdash \texttt{type}(\tau)\ \mathsf{tsmty}$ by Rule (C.25a) on (4)

Case (C.22b).

(1) $\hat{\rho} = \forall \hat{X}{:}\hat{\sigma}.\hat{\rho}'$ by assumption

(2) $\rho = \texttt{allmods}\{\sigma\}(X.\rho')$ by assumption

(3) $\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma\ \mathsf{sig}$ by assumption

(4) $\hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma \vdash \hat{\rho}' \rightsquigarrow \rho'\ \mathsf{tsmty}$ by assumption

(5) $\Omega \vdash \sigma\ \mathsf{sig}$ by Theorem C.36 on (3)

(6) $\Omega, X : \sigma \vdash \rho'\ \mathsf{tsmty}$ by IH on (4)

(7) $\Omega \vdash \texttt{allmods}\{\sigma\}(X.\rho')\ \mathsf{tsmty}$ by Rule (C.25b) on (5) and (6)

□

Theorem C.38 (peTSM Expression Expansion). *If* $\langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle \vdash^{\mathsf{Exp}}_{\langle \mathcal{A}; \Psi \rangle} \hat{e} \rightsquigarrow \epsilon \mathbin{@} \rho$ *then* $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} \rho$.

Proof. By rule induction over Rules (C.23). In the following, let $\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle$ and $\hat{\Psi} = \langle \mathcal{A}; \Psi \rangle$.

Case (C.23a).

(1) $\hat{e} = \hat{a}$ by assumption

(2) $\mathcal{A} = \mathcal{A}', \hat{a} \hookrightarrow \epsilon$ by assumption

(3) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} \rho$ by assumption

Case (C.23b).

(1) $\hat{e} = \Lambda \hat{X}{:}\hat{\sigma}.\hat{e}'$ by assumption

(2) $\epsilon = \texttt{absmod}\{\sigma\}(X.\epsilon')$ by assumption

(3) $\rho = \texttt{allmods}\{\sigma\}(X.\rho')$ by assumption

(4) $\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma\ \mathsf{sig}$ by assumption

(5) $\hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{e}' \rightsquigarrow \epsilon' \mathbin{@} \rho'$ by assumption

(6) $\Omega, X : \sigma \vdash^{\mathsf{Exp}}_{\Psi} \epsilon' \mathbin{@} \rho'$ by IH on (4) and (5)

(7) $\Omega \vdash \sigma\ \mathsf{sig}$ by Theorem C.36 on (4)

(8) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \texttt{absmod}\{\sigma\}(X.\epsilon') \mathbin{@} \texttt{allmods}\{\sigma\}(X.\rho')$ by Rule (C.26b) on (7) and (6)

Case (C.23c).

(1) $\hat{e} = \hat{e}'(\hat{X})$ by assumption

(2) $\epsilon = \texttt{apmod}\{X\}(\epsilon')$ by assumption

(3) $\rho = [X/X']\rho'$ by assumption

(4) $\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{e}' \rightsquigarrow \epsilon' \mathbin{@} \texttt{allmods\{}\sigma\texttt{\}(}X'.\rho'\texttt{)}$ by assumption

(5) $\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{X} \rightsquigarrow X : \sigma$ by assumption

(6) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon' \mathbin{@} \texttt{allmods\{}\sigma\texttt{\}(}X'.\rho'\texttt{)}$ by IH on (4)

(7) $\Omega \vdash X : \sigma$ by Theorem C.44 on (5)

(8) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \texttt{apmod\{}X\texttt{\}(}\epsilon'\texttt{)} \mathbin{@} [X/X']\rho'$ by Rule (C.26c) on (6) and (7)

□

Theorem C.39 (ppTSM Expression Expansion). *If $\langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle \vdash^{\mathsf{Pat}}_{\langle \mathcal{A};\Phi \rangle} \hat{e} \rightsquigarrow \epsilon \mathbin{@} \rho$ then $\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mathbin{@} \rho$.*

Proof. The proof is nearly identical to the proof of Theorem C.38, differing only in that ppTSM contexts and the corresponding judgements are mentioned. □

Patterns

Lemma C.40 (Proto-Pattern Deparameterization). *If $\Omega_{app} \vdash^{\mathsf{Pat}}_{\Phi} \dot{p} \looparrowright_{\epsilon} \grave{p} \mathbin{?} \rho \dashv \omega : \Omega_{params}$ then $dom(\Omega_{app}) \cap dom(\Omega_{params}) = \emptyset$ and $\Omega_{app} \vdash \omega : \Omega_{params}$ and $\Omega_{app} \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mathbin{@} [\omega]\rho$.*

Proof. By rule induction over Rules (C.38).

Case (C.38a). We have:

(1) $\epsilon = \texttt{defref[}a\texttt{]}$ by assumption

(2) $\omega = \emptyset$ by assumption

(3) $\Phi = \Phi', a \hookrightarrow \texttt{pptsm}(\rho; e_{\text{parse}})$ by assumption

(4) $\Omega_{\text{params}} = \emptyset$ by assumption

(5) $\Omega_{\text{app}} \vdash \rho\ \mathsf{tsmty}$ by assumption

(6) $\text{dom}(\Omega_{\text{app}}) \cap \text{dom}(\emptyset) = \emptyset$ by definition

(7) $\Omega_{\text{app}} \vdash \emptyset : \emptyset$ by definition

(8) $[\emptyset]\rho = \rho$ by definition

(9) $\Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi', a \hookrightarrow \texttt{pptsm}(\rho; e_{\text{parse}})} a \mathbin{@} \rho$ by Rule (C.27a) on (5)

Case (C.38b). We have:

(1) $\epsilon = \texttt{apmod\{}X\texttt{\}(}\epsilon'\texttt{)}$ by assumption

(2) $\Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi} \dot{p} \looparrowright_{\epsilon'} \grave{p} \mathbin{?} \texttt{allmods\{}\sigma\texttt{\}(}X'.\rho\texttt{)} \dashv \omega' : \Omega'$ by assumption

(3) $\Omega_{\text{app}} \vdash X : \sigma$ by assumption

(4) $X' \notin \text{dom}(\Omega_{\text{app}})$ by assumption

(5) $\omega = \omega', X/X'$ by assumption

(6) $\Omega_{\text{params}} = \Omega', X' : \sigma$ by assumption

(7) $\text{dom}(\Omega_{\text{app}}) \cap \text{dom}(\Omega') = \emptyset$ by IH on (2)

(8) $\Omega_{\text{app}} \vdash \omega' : \Omega'$ by IH on (2)

(9) $\Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi} \epsilon' \mathbin{@} [\omega']\texttt{allmods\{}\sigma\texttt{\}(}X'.\rho\texttt{)}$ by IH on (2)

(10) $\mathsf{dom}(\Omega_{\mathsf{app}}) \cap \mathsf{dom}(\Omega', X' : \sigma)$ by (4) and (7) and definition of finite set intersection

(11) $\Omega_{\mathsf{app}} \vdash \omega', X/X' : \Omega', X' : \sigma$ by Definition C.2 on (3)

(12) $\Omega_{\mathsf{app}} \vdash^{\mathsf{Pat}}_{\Phi} \mathtt{apmod\{}X\mathtt{\}(}\epsilon'\mathtt{)} \mathbin{@} (\omega', X/X')\rho$ by Rule (C.27c) on (9) and (3)

□

Theorem C.41 (Typed Pattern Expansion)**.**

1. *If $\langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{app} \rangle \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\!| \langle \mathcal{M}'; \mathcal{D}'; \mathcal{G}'; \Omega' \rangle$ then $\mathcal{M}' = \emptyset$ and $\mathcal{D}' = \emptyset$ and $\Omega_{app} \vdash p : \tau \dashv\!\!| \Omega'$.*
2. *If $\grave{p} \rightsquigarrow p : \tau \dashv\!\!|^{\omega:\Omega_{params};\, \langle \mathcal{M};\mathcal{D};\mathcal{G};\Omega_{app}\rangle;\, \hat{\Phi};\, b} \langle \mathcal{M}'; \mathcal{D}'; \mathcal{G}'; \Omega' \rangle$ and $dom(\Omega_{params}) \cap dom(\Omega_{app}) = \emptyset$ then $\mathcal{M}' = \emptyset$ and $\mathcal{D}' = \emptyset$ and $\Omega_{params} \cup \Omega_{app} \vdash p : \tau \dashv\!\!| \Omega'$.*

Proof. My mutual rule induction over Rules (C.21) and Rules (C.43).

1. In the following, let $\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\mathsf{app}} \rangle$ and $\hat{\Omega}' = \langle \mathcal{M}'; \mathcal{D}'; \mathcal{G}'; \Omega' \rangle$.

   Case (C.21a) **through** (C.21f)**.** These cases follow by applying the IH, part 1 and applying the corresponding pattern typing rule in Rules (C.14).

   Case (C.21g)**.** We have:

   (1) $\hat{p} = \hat{e}\ \text{`}b\text{`}$ by assumption

   (2) $\hat{\Phi} = \langle \mathcal{A}; \Phi \rangle$ by assumption

   (3) $\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \hat{e} \rightsquigarrow \epsilon \mathbin{@} \mathtt{type(}\tau_{\mathsf{final}}\mathtt{)}$ by assumption

   (4) $\Omega_{\mathsf{app}} \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \Downarrow \epsilon_{\mathsf{normal}}$ by assumption

   (5) $\mathsf{tsmdef}(\epsilon_{\mathsf{normal}}) = a$ by assumption

   (6) $\Phi = \Phi', a \hookrightarrow \mathtt{pptsm}(\rho; e_{\mathsf{parse}})$ by assumption

   (7) $b \downarrow_{\mathsf{Body}} e_{\mathsf{body}}$ by assumption

   (8) $e_{\mathsf{parse}}(e_{\mathsf{body}}) \Downarrow \mathtt{inj[SuccessP]}(e_{\mathsf{proto}})$ by assumption

   (9) $e_{\mathsf{proto}} \uparrow_{\mathsf{PPrPat}} \dot{p}$ by assumption

   (10) $\Omega_{\mathsf{app}} \vdash^{\mathsf{Pat}}_{\Phi} \dot{p} \looparrowright_{\epsilon_{\mathsf{normal}}} \grave{p}\ ?\ \mathtt{type(}\tau_{\mathsf{proto}}\mathtt{)} \dashv \omega : \Omega_{\mathsf{params}}$ by assumption

   (11) $\grave{p} \rightsquigarrow p : \tau_{\mathsf{proto}} \dashv\!\!|^{\omega:\Omega_{\mathsf{params}};\, \hat{\Omega};\, \hat{\Phi};\, b} \hat{\Omega}'$ by assumption

   (12) $\tau = [\omega]\tau_{\mathsf{proto}}$ by assumption

   (13) $\mathsf{dom}(\Omega_{\mathsf{params}}) \cap \mathsf{dom}(\Omega_{\mathsf{app}}) = \emptyset$ by Lemma C.40 on (10)

   (14) $\Omega_{\mathsf{app}} \vdash \omega : \Omega_{\mathsf{params}}$ by Lemma C.40 on (10)

   (15) $\mathcal{M}' = \emptyset$ and $\mathcal{D}' = \emptyset$ by IH, part 2 on (11) and (13)

   (16) $\Omega_{\mathsf{params}} \cup \Omega_{\mathsf{app}} \vdash p : \tau_{\mathsf{proto}} \dashv\!\!| \Omega'$ by IH, part 2 on (11) and (13)

   (17) $\Omega_{\mathsf{app}} \vdash p : [\omega]\tau_{\mathsf{proto}} \dashv\!\!| \Omega'$ by Substitution Lemma C.3 on (14) and (16)

2. We induct on the premise. In the following, let $\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\text{app}} \rangle$ and $\hat{\Omega}' = \langle \mathcal{M}'; \mathcal{D}'; \mathcal{G}'; \Omega' \rangle$.

Case (C.43a) **through** (C.43d)**.** These cases follow by applying the IH, part 2 and then applying the corresponding pattern rule in Rules (C.14).

Case (C.43e)**.**

(1) $\grave{p} = \texttt{splicedp}[m; n; \grave{\tau}]$	by assumption
(2) $\tau = [\omega]\tau'$	by assumption
(3) $\Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}};\, \hat{\Omega};\, b} \grave{\tau} \rightsquigarrow \tau' :: \texttt{Type}$	by assumption
(4) $\mathsf{parseUPat}(\mathsf{subseq}(b; m; n)) = \grave{p}$	by assumption
(5) $\hat{\Omega} \vdash_{\hat{\Phi}} \grave{p} \rightsquigarrow p : [\omega]\tau' \dashv\!\vert\, \hat{\Omega}'$	by assumption
(6) $\mathcal{M}' = \emptyset$ and $\mathcal{D}' = \emptyset$	by IH, part 1 on (5)
(7) $\Omega_{\text{app}} \vdash p : [\omega]\tau' \dashv\!\vert\, \Omega'$	by IH, part 1 on (5)
(8) $\Omega_{\text{params}} \cup \Omega_{\text{app}} \vdash p : [\omega]\tau' \dashv\!\vert\, \Omega'$	by Weakening on (7)

The mutual induction can be shown to be well-founded by an argument analagous to that in the proof of Theorem B.26, appealing to Condition C.14 and Condition C.16.

□

Expressions and Rules

Lemma C.42 (Proto-Expression Deparameterization)**.** *If $\Omega_{app} \vdash^{\mathsf{Exp}}_{\Psi} \dot{e} \looparrowright_{\epsilon} \grave{e} \mathrel{?} \rho \dashv \omega : \Omega_{params}$ then $dom(\Omega_{app}) \cap dom(\Omega_{params}) = \emptyset$ and $\Omega_{app} \vdash \omega : \Omega_{params}$ and $\Omega_{app} \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} [\omega]\rho$.*

Proof. By rule induction over Rules (C.37).

Case (C.37a)**.** We have:

(1) $\epsilon = \texttt{defref}[a]$	by assumption
(2) $\omega = \emptyset$	by assumption
(3) $\Psi = \Psi', a \hookrightarrow \texttt{petsm}(\rho; e_{\text{parse}})$	by assumption
(4) $\Omega_{\text{params}} = \emptyset$	by assumption
(5) $\Omega_{\text{app}} \vdash \rho\ \mathsf{tsmty}$	by assumption
(6) $\mathrm{dom}(\Omega_{\text{app}}) \cap \mathrm{dom}(\emptyset) = \emptyset$	by definition
(7) $\Omega_{\text{app}} \vdash \emptyset : \emptyset$	by definition
(8) $[\emptyset]\rho = \rho$	by definition
(9) $\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi', a \hookrightarrow \texttt{petsm}(\rho; e_{\text{parse}})} a \mathbin{@} \rho$	by Rule (C.26a) on (5)

Case (C.37b)**.** We have:

(1) $\epsilon = \texttt{apmod\{}X\texttt{\}(}\epsilon'\texttt{)}$	by assumption
(2) $\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \dot{e} \looparrowright_{\epsilon'} \grave{e} \mathrel{?} \texttt{allmods\{}\sigma\texttt{\}(}X'.\rho\texttt{)} \dashv \omega' : \Omega'$	by assumption
(3) $\Omega_{\text{app}} \vdash X : \sigma$	by assumption
(4) $X' \notin \mathrm{dom}(\Omega_{\text{app}})$	by assumption
(5) $\omega = \omega', X/X'$	by assumption
(6) $\Omega_{\text{params}} = \Omega', X' : \sigma$	by assumption
(7) $\mathrm{dom}(\Omega_{\text{app}}) \cap \mathrm{dom}(\Omega') = \emptyset$	by IH on (2)

(8) $\Omega_{\text{app}} \vdash \omega' : \Omega'$ by IH on (2)

(9) $\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \epsilon' \mathbin{@} [\omega']\texttt{allmods\{}\sigma\texttt{\}(}X'.\rho\texttt{)}$ by IH on (2)

(10) $\text{dom}(\Omega_{\text{app}}) \cap \text{dom}(\Omega', X' : \sigma)$ by (4) and (7) and definition of finite set intersection

(11) $\Omega_{\text{app}} \vdash \omega', X/X' : \Omega', X' : \sigma$ by Definition C.2 on (3)

(12) $\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \texttt{apmod\{}X\texttt{\}(}\epsilon'\texttt{)} \mathbin{@} (\omega', X/X')\rho$ by Rule (C.26c) on (9) and (3)

$\square$

Theorem C.43 (Typed Expression and Rule Expansion)**.**

1. *(a) If* $\langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau$ *then* $\Omega \vdash e : \tau$.

   (b) If $\langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \rightsquigarrow r : \tau \Mapsto \tau'$ *then* $\Omega \vdash r : \tau \Mapsto \tau'$.

2. *(a) If* $\Omega \vdash^{\omega:\Omega_{params}; \langle \mathcal{M};\mathcal{D};\mathcal{G};\Omega_{app}\rangle; \hat{\Psi}; \hat{\Phi}; b} \grave{e} \rightsquigarrow e : \tau$ *and* $dom(\Omega) \cap dom(\Omega_{app}) = \emptyset$ *then* $\Omega \cup \Omega_{app} \vdash e : \tau$.

   (b) If $\Omega \vdash^{\omega:\Omega_{params}; \langle \mathcal{M};\mathcal{D};\mathcal{G};\Omega_{app}\rangle; \hat{\Psi}; \hat{\Phi}; b} \grave{r} \rightsquigarrow r : \tau \Mapsto \tau'$ *and* $dom(\Omega) \cap dom(\Omega_{app}) = \emptyset$ *then* $\Omega \cup \Omega_{app} \vdash r : \tau \Mapsto \tau'$.

Proof. By mutual rule induction over Rules (C.19), Rule (C.20), Rules (C.41) and Rule (C.42).

1. (a) In the following, let $\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle$.

Case (C.19a)**.**

(1) $\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau'$ by assumption

(2) $\Omega \vdash \tau <: \tau'$ by assumption

(3) $\Omega \vdash e : \tau'$ by IH, part 1(a) on (1)

(4) $\Omega \vdash e : \tau$ by Rule (C.12a) on (3) and (2)

Case (C.19b) **through** (C.19o)**.** In each of these cases, we apply the IH, part 1(a) or 1(b), over the premises and then apply the corresponding typing rule in Rules (C.12) and weakening as needed.

Case (C.19p)**.**

(1) $\hat{e} = \hat{e}$ ‘b‘ by assumption

(2) $e = [\omega]e'$ by assumption

(3) $\tau = [\omega]\tau_{\text{proto}}$ by assumption

(4) $\hat{\Psi} = \langle \mathcal{A}; \Psi \rangle$ by assumption

(5) $\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{e} \rightsquigarrow \epsilon \mathbin{@} \texttt{type(}\tau_{\text{final}}\texttt{)}$ by assumption

(6) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \Downarrow \epsilon_{\text{normal}}$ by assumption

(7) $\mathsf{tsmdef}(\epsilon_{\text{normal}}) = a$ by assumption

(8) $\Psi = \Psi', a \hookrightarrow \texttt{petsm}(\rho; e_{\text{parse}})$ by assumption

(9) $b \downarrow_{\mathsf{Body}} e_{\text{body}}$ by assumption

(10) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj[SuccessE](}e_{\text{pproto}}\texttt{)}$ by assumption

(11) $e_{\text{pproto}} \uparrow_{\mathsf{PPrExpr}} \dot{e}$ by assumption

(12) $\Omega \vdash^{\mathsf{Exp}}_{\Psi} \dot{e} \looparrowright_{\epsilon_{\text{normal}}} \grave{e}\ ?\ \mathtt{type}(\tau_{\text{proto}}) \dashv \omega : \Omega_{\text{params}}$ by assumption

(13) $\Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}};\hat{\Omega};\hat{\Psi};\hat{\Phi};b} \grave{e} \leadsto e' : \tau_{\text{proto}}$ by assumption

(14) $\Omega \vdash \omega : \Omega_{\text{params}}$ by Lemma C.42 on (12)

(15) $\Omega \cup \Omega_{\text{params}} \vdash e' : \tau_{\text{proto}}$ by IH, part 2(a) on (13)

(16) $\Omega \vdash [\omega]e' : [\omega]\tau_{\text{proto}}$ by Substitution Lemma C.3 on (14) and (15)

(b) In the following, let $\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle$.

Case (C.20).

(1) $\hat{r} = \hat{p} \Rightarrow \hat{e}$ by assumption

(2) $r = \mathtt{rule}(p.e)$ by assumption

(3) $\hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \leadsto p : \tau \dashv\!\mid \langle \emptyset; \emptyset; \mathcal{G}'; \Omega' \rangle$ by assumption

(4) $\langle \mathcal{M}; \mathcal{D}; \mathcal{G} \uplus \mathcal{G}'; \Omega \cup \Omega' \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e : \tau'$ by assumption

(5) $\Omega \vdash p : \tau \dashv\!\mid \Omega'$ by Theorem C.41 on (3)

(6) $\Omega \cup \Omega' \vdash e : \tau'$ by IH, part 1(a) on (4)

(7) $\Omega \vdash r : \tau \Mapsto \tau'$ by Rule (C.13) on (5) and (6)

2. (a) In the following, let $\mathbb{E} = \omega : \Omega_{\text{params}}; \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\text{app}} \rangle; \hat{\Psi}; \hat{\Phi}; b$.

Case (C.41a).

(1) $\Omega \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau'$ by assumption

(2) $\Omega \vdash \tau' <: \tau$ by assumption

(3) $\Omega \cup \Omega_{\text{app}} \vdash e : \tau'$ by IH, part 2(a) on (1)

(4) $\Omega \cup \Omega_{\text{app}} \vdash e : \tau$ by Rule (C.12a) on (3) and (2)

Case (C.41b) **through** (C.41o). In each of these cases, we apply the IH, part 2(a) or 2(b), over the premises and then apply the corresponding typing rule in Rules (C.12) and weakening as needed.

Case (C.41p).

(1) $\grave{e} = \mathtt{splicede}[m; n; \grave{\tau}]$ by assumption

(2) $\tau = [\omega]\tau'$ by assumption

(3) $\Omega_{\text{params}} \vdash^{\mathsf{cs}(\mathbb{E})} \grave{\tau} \leadsto \tau' :: \mathtt{Type}$ by assumption

(4) $\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e}$ by assumption

(5) $\langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\text{app}} \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e : [\omega]\tau'$ by assumption

(6) $\Omega_{\text{app}} \vdash e : [\omega]\tau'$ by IH, part 1(a) on (5)

(7) $\Omega_{\text{app}} \cup \Omega \vdash e : [\omega]\tau'$ by Weakening on (6)

(b) **Case** (C.42).

(1) $\grave{r} = \texttt{prrule}(p.\grave{e})$ by assumption

(2) $r = \texttt{rule}(p.e)$ by assumption

(3) $\Omega \vdash p : \tau \dashv\!\!\mid \Omega'$ by assumption

(4) $\Omega \cup \Omega' \vdash^{\omega:\Omega_{\text{params}};\langle\mathcal{M};\mathcal{D};\mathcal{G};\Omega_{\text{app}}\rangle;\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e : \tau'$

by assumption

(5) $(\Omega \cup \Omega') \cap \Omega_{\text{app}} = \emptyset$ by identification convention

(6) $\Omega \cup \Omega' \cup \Omega_{\text{app}} \vdash e : \tau'$ by IH, part 2(a) on (4) and (5)

(7) $\Omega \cup \Omega_{\text{app}} \vdash p : \tau \dashv\!\!\mid \Omega'$ by Weakening on (3)

(8) $\Omega \cup \Omega_{\text{app}} \vdash r : \tau \Mapsto \tau'$ by Rule (C.13) on (7) and (6)

The mutual induction can be shown to be well-founded by an argument analagous to that in the proof of Theorem B.27, appealing to Condition C.13 and Condition C.16. □

Modules

Theorem C.44 (Module Expansion). *If $\langle\mathcal{M};\mathcal{D};\mathcal{G};\Omega\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma$ then $\Omega \vdash M : \sigma$.*

Proof. By rule induction over Rules (C.16). In the following, let $\hat{\Omega} = \langle\mathcal{M};\mathcal{D};\mathcal{G};\Omega\rangle$.

Case (C.16a)**.**

(1) $\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{M} \rightsquigarrow M : \sigma'$ by assumption

(2) $\hat{\Omega} \vdash \sigma' <: \sigma$ by assumption

(3) $\Omega \vdash M : \sigma'$ by IH on (1)

(4) $\Omega \vdash M : \sigma$ by Rule (C.4a) on (3) and (2)

Case (C.16b) **through** (C.16e)**.** In each of these cases, we apply the IH over each module expansion premise, Theorem C.43 over each expression expansion premise and Theorem C.35 over each construction expansion premise, then apply the corresponding signature matching rule in Rules (C.4) and weakening as needed.

Case (C.16f) **through** (C.16i)**.** In each of these cases, we apply the IH to the module expansion premise.

□

C.4.3 Abstract Reasoning Principles

Lemma C.45 (Proto-Construction and Proto-Kind Decomposition)**.**

1. *If $\Omega \vdash^{\omega:\Omega_{params};\hat{\Omega};b} \grave{c} \rightsquigarrow c :: \kappa$ where $\mathsf{summary}(\grave{c}) = \{\texttt{splicedk}[m_i;n_i]\}_{0\leq i<n_{kind}} \cup \{\texttt{splicedc}[m'_i;n'_i;\grave{\kappa}'_i]\}_{0\leq i<n_{con}}$ then*
   (a) $\{\hat{\Omega} \vdash \mathsf{parseUKind}(\mathsf{subseq}(b;m_i;n_i)) \rightsquigarrow \kappa_i\ \mathsf{kind}\}_{0\leq i<n_{kind}}$
   (b) $\{\Omega_{params} \vdash^{\omega:\Omega_{params};\hat{\Omega};b} \grave{\kappa}'_i \rightsquigarrow \kappa'_i\ \mathsf{kind}\}_{0\leq i<n_{con}}$
   (c) $\{\hat{\Omega} \vdash \mathsf{parseUCon}(\mathsf{subseq}(b;m'_i;n'_i)) \rightsquigarrow c_i :: [\omega]\kappa'_i\}_{0\leq i<n_{con}}$

(d) $c = [\{\kappa_i/k_i\}_{0\leq i<n_{kind}}, \{c_i/u_i\}_{0\leq i<n_{con}}, \omega]c'$ *for some* c' *and fresh* $\{k_i\}_{0\leq i<n_{kind}}$ *and fresh* $\{u_i\}_{0\leq i<n_{con}}$

(e) $\mathsf{fv}(c') \subset \{k_i\}_{0\leq i<n_{kind}} \cup \{u_i\}_{0\leq i<n_{con}} \cup dom(\Omega)$

2. *If* $\Omega \vdash^{\omega:\Omega_{params};\hat{\Omega};b} \grave{\kappa} \rightsquigarrow \kappa\ \mathsf{kind}$ *where* $\mathsf{summary}(\grave{\kappa}) = \{\texttt{splicedk}[m_i;n_i]\}_{0\leq i<n_{kind}} \cup \{\texttt{splicedc}[m_i';n_i';\grave{\kappa}_i']\}_{0\leq i<n_{con}}$ *then*

(a) $\{\hat{\Omega} \vdash \mathsf{parseUKind}(\mathsf{subseq}(b;m_i;n_i)) \rightsquigarrow \kappa_i\ \mathsf{kind}\}_{0\leq i<n_{kind}}$

(b) $\{\Omega_{params} \vdash^{\omega:\Omega_{params};\hat{\Omega};b} \grave{\kappa}_i' \rightsquigarrow \kappa_i'\ \mathsf{kind}\}_{0\leq i<n_{con}}$

(c) $\{\hat{\Omega} \vdash \mathsf{parseUCon}(\mathsf{subseq}(b;m_i';n_i')) \rightsquigarrow c_i :: [\omega]\kappa_i'\}_{0\leq i<n_{con}}$

(d) $\kappa = [\{\kappa_i/k_i\}_{0\leq i<n_{kind}}, \{c_i/u_i\}_{0\leq i<n_{con}}, \omega]\kappa'$ *for some* κ' *and fresh* $\{k_i\}_{0\leq i<n_{kind}}$ *and fresh* $\{u_i\}_{0\leq i<n_{con}}$

(e) $\mathsf{fv}(\kappa') \subset \{k_i\}_{0\leq i<n_{kind}} \cup \{u_i\}_{0\leq i<n_{con}} \cup dom(\Omega)$

Proof. By mutual rule induction over Rules (C.40) and Rules (C.39).

1. **Case** (C.40a)**.** This case follows by applying the IH.

Case (C.40b) **through** (C.40o)**.** These cases follow by applying the IH, gathering together the sets of conclusions and invoking the identification convention as necessary.

Case (C.40p)**.** Letting $\mathbb{C} = \omega : \Omega_{\text{params}};\ \hat{\Omega};\ b$,

(1) $\grave{c} = \texttt{splicedc}[m;n;\grave{\kappa}]$ by assumption

(2) $\Omega_{\text{params}} \vdash^{\mathbb{C}} \grave{\kappa} \rightsquigarrow \kappa\ \mathsf{kind}$ by assumption

(3) $\mathsf{parseUCon}(\mathsf{subseq}(b;m;n)) = \hat{c}$ by assumption

(4) $\hat{\Omega} \vdash \hat{c} \rightsquigarrow c :: [\omega]\kappa$ by assumption

(5) $\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\text{app}} \rangle$ by assumption

(6) $\text{dom}(\Omega) \cap \text{dom}(\Omega_{\text{app}}) = \emptyset$ by assumption

(7) $\mathsf{summary}(\texttt{splicedc}[m;n;\grave{\kappa}]) = \mathsf{summary}(\grave{\kappa}) \cup \{\texttt{splicedc}[m;n;\grave{\kappa}]\}$ by definition

(8) $\mathsf{summary}(\grave{\kappa}) = \{\texttt{splicedk}[m_i;n_i]\}_{0\leq i<n_{\text{kind}}} \cup \{\texttt{splicedc}[m_i';n_i';\grave{\kappa}_i']\}_{0\leq i<n_{\text{con}}-1}$ by definition

(9) $\{\hat{\Omega} \vdash \mathsf{parseUKind}(\mathsf{subseq}(b;m_i;n_i)) \rightsquigarrow \kappa_i\ \mathsf{kind}\}_{0\leq i<n_{\text{kind}}}$ by IH, part 2 on (2) and (8)

(10) $\{\Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}};\hat{\Omega};b} \grave{\kappa}_i' \rightsquigarrow \kappa_i'\ \mathsf{kind}\}_{0\leq i<n_{\text{con}}-1}$ by IH, part 2 on (2) and (8)

(11) $\{\hat{\Omega} \vdash \mathsf{parseUCon}(\mathsf{subseq}(b;m_i';n_i')) \rightsquigarrow c_i :: [\omega]\kappa_i'\}_{0\leq i<n_{\text{con}}-1}$ by IH, part 2 on (2) and (8)

The conclusions hold as follows:

(a) (9)

(b) (2) and (10)

(c) (3), (4) and (11)

(d) Choose $c' = u$ for fresh u. Then $c = [\omega', c/u]u$ for any ω'.

(e) $\mathsf{fv}(u) = u$ and $u \subset \{u\} \cup \{k_i\}_{0 \leq i < n_{\text{kind}}} \cup \{u_i\}_{0 \leq i < n_{\text{con}}-1} \cup \text{dom}(\Omega)$ by definition.

2. **Case** (C.39a) **through** (C.39e)**.** These cases follow by applying the IH and gathering together the sets of conclusions, invoking the identification convention as necessary.

Case (C.39f)**.** Letting $\mathbb{C} = \omega : \Omega_{\text{params}}; \hat{\Omega}; b$

(1) $\grave{\kappa} = \texttt{splicedk}[m; n]$	by assumption
(2) $\mathsf{parseUKind}(\mathsf{subseq}(b; m; n)) = \hat{\kappa}$	by assumption
(3) $\hat{\Omega} \vdash \hat{\kappa} \rightsquigarrow \kappa \ \mathsf{kind}$	by assumption
(4) $\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\text{app}} \rangle$	by assumption
(5) $\text{dom}(\Omega) \cap \text{dom}(\Omega_{\text{app}}) = \emptyset$	by assumption
(6) $n_{\text{con}} = 0$	by definition of summary

The conclusions hold as follows:

(a) (2) and (3)

(b) (6)

(c) (6)

(d) Choose $\kappa' = k$ for fresh k. Then $\kappa = [\omega', \kappa/k]k$ for any ω'.

(e) $\mathsf{fv}(k) = k$ and $k \subset \{k\} \cup \text{dom}(\Omega)$ by definition.

$\square$

Lemma C.46 (Proto-Expression and Proto-Rule Decomposition)**.**

1. *If* $\Omega \vdash^{\omega:\Omega_{params}; \hat{\Omega}; \hat{\Psi}; \hat{\Phi}; b} \grave{e} \rightsquigarrow e : \tau_{proto}$ *where* $\mathsf{summary}(\grave{e}) = \{\texttt{splicedk}[m_i; n_i]\}_{0 \leq i < n_{kind}} \cup \{\texttt{splicedc}[m_i'; n_i'; \grave{\kappa}_i']\}_{0 \leq i < n_{con}} \cup \{\texttt{splicede}[m_i''; n_i''; \grave{\tau}_i]\}_{0 \leq i < n_{exp}}$ *then*
   (a) $\{\hat{\Omega} \vdash \mathsf{parseUKind}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow \kappa_i \ \mathsf{kind}\}_{0 \leq i < n_{kind}}$
   (b) $\{\Omega_{params} \vdash^{\omega:\Omega_{params}; \hat{\Omega}; b} \grave{\kappa}_i' \rightsquigarrow \kappa_i' \ \mathsf{kind}\}_{0 \leq i < n_{con}}$
   (c) $\{\hat{\Omega} \vdash \mathsf{parseUCon}(\mathsf{subseq}(b; m_i'; n_i')) \rightsquigarrow c_i :: [\omega]\kappa_i'\}_{0 \leq i < n_{con}}$
   (d) $\{\Omega_{params} \vdash^{\omega:\Omega_{params}; \hat{\Omega}; b} \grave{\tau}_i \rightsquigarrow \tau_i :: \texttt{Type}\}_{0 \leq i < n_{exp}}$
   (e) $\{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \mathsf{parseUExp}(\mathsf{subseq}(b; m_i''; n_i'')) \rightsquigarrow e_i : [\omega]\tau_i\}_{0 \leq i < n_{exp}}$
   (f) $e = [\{\kappa_i/k_i\}_{0 \leq i < n_{kind}}, \{c_i/u_i\}_{0 \leq i < n_{con}}, \{e_i/x_i\}_{0 \leq i < n_{exp}}, \omega]e''$ *for some* e'' *and fresh* $\{k_i\}_{0 \leq i < n_{kind}}$ *and fresh* $\{u_i\}_{0 \leq i < n_{con}}$ *and fresh* $\{x_i\}_{0 \leq i < n_{exp}}$
   (g) $\mathsf{fv}(e'') \subset \{k_i\}_{0 \leq i < n_{kind}} \cup \{u_i\}_{0 \leq i < n_{con}} \cup \{x_i\}_{0 \leq i < n_{exp}} \cup \textit{dom}(\Omega)$
2. *If* $\Omega \vdash^{\omega:\Omega_{params}; \hat{\Omega}; \hat{\Psi}; \hat{\Phi}; b} \grave{r} \rightsquigarrow r : \tau \Mapsto \tau'$ *where* $\mathsf{summary}(\grave{r}) = \{\texttt{splicedk}[m_i; n_i]\}_{0 \leq i < n_{kind}} \cup \{\texttt{splicedc}[m_i'; n_i'; \grave{\kappa}_i']\}_{0 \leq i < n_{con}} \cup \{\texttt{splicede}[m_i''; n_i''; \grave{\tau}_i]\}_{0 \leq i < n_{exp}}$ *then*
   (a) $\{\hat{\Omega} \vdash \mathsf{parseUKind}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow \kappa_i \ \mathsf{kind}\}_{0 \leq i < n_{kind}}$
   (b) $\{\Omega_{params} \vdash^{\omega:\Omega_{params}; \hat{\Omega}; b} \grave{\kappa}_i' \rightsquigarrow \kappa_i' \ \mathsf{kind}\}_{0 \leq i < n_{con}}$
   (c) $\{\hat{\Omega} \vdash \mathsf{parseUCon}(\mathsf{subseq}(b; m_i'; n_i')) \rightsquigarrow c_i :: [\omega]\kappa_i'\}_{0 \leq i < n_{con}}$
   (d) $\{\Omega_{params} \vdash^{\omega:\Omega_{params}; \hat{\Omega}; b} \grave{\tau}_i \rightsquigarrow \tau_i :: \texttt{Type}\}_{0 \leq i < n_{exp}}$
   (e) $\{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \mathsf{parseUExp}(\mathsf{subseq}(b; m_i''; n_i'')) \rightsquigarrow e_i : [\omega]\tau_i\}_{0 \leq i < n_{exp}}$

(f) $r = [\{\kappa_i/k_i\}_{0\leq i<n_{kind}}, \{c_i/u_i\}_{0\leq i<n_{con}}, \{e_i/x_i\}_{0\leq i<n_{exp}}, \omega]r'$ *for some* r' *and fresh* $\{k_i\}_{0\leq i<n_{kind}}$ *and fresh* $\{u_i\}_{0\leq i<n_{con}}$ *and fresh* $\{x_i\}_{0\leq i<n_{exp}}$

(g) $\mathsf{fv}(r') \subset \{k_i\}_{0\leq i<n_{kind}} \cup \{u_i\}_{0\leq i<n_{con}} \cup \{x_i\}_{0\leq i<n_{exp}} \cup dom(\Omega)$

Proof. By mutual rule induction over Rules (C.41) and Rule (C.42).

1. **Case** (C.41a)**.** This case follows by applying the IH.

   Case (C.41b) **through** (C.41o)**.** These cases follow by applying the IH or Lemma C.45, gathering together the sets of conclusions and invoking the identification convention as necessary.

   Case (C.41p)**.** Letting $\mathbb{E} = \omega : \Omega_{\text{params}}; \hat{\Omega}; \hat{\Psi}; \hat{\Phi}; b$,

   (1) $\grave{e} = \texttt{splicede}[m; n; \grave{\tau}]$ — by assumption
   (2) $\Omega_{\text{params}} \vdash^{\mathsf{cs}(\mathbb{E})} \grave{\tau} \rightsquigarrow \tau :: \texttt{Type}$ — by assumption
   (3) $\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e}$ — by assumption
   (4) $\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : [\omega]\tau$ — by assumption
   (5) $\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\text{app}} \rangle$ — by assumption
   (6) $\text{dom}(\Omega) \cap \text{dom}(\Omega_{\text{app}}) = \emptyset$ — by assumption
   (7) $\mathsf{summary}(\grave{e}) = \mathsf{summary}(\grave{\tau}) \cup \{\texttt{splicede}[m; n; \grave{\tau}]\}$ — by assumption
   (8) $\mathsf{summary}(\grave{\tau}) = \{\texttt{splicedk}[m_i; n_i]\}_{0\leq i<n_{\text{kind}}} \cup \{\texttt{splicedc}[m'_i; n'_i; \grave{\kappa}'_i]\}_{0\leq i<n_{\text{con}}}$ — by definition
   (9) $\{\hat{\Omega} \vdash \mathsf{parseUKind}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow \kappa_i\ \mathsf{kind}\}_{0\leq i<n_{\text{kind}}}$ — by Lemma C.45 on (2) and (8)
   (10) $\{\Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}};\hat{\Omega};b} \grave{\kappa}'_i \rightsquigarrow \kappa'_i\ \mathsf{kind}\}_{0\leq i<n_{\text{con}}}$ — by Lemma C.45 on (2) and (8)
   (11) $\{\hat{\Omega} \vdash \mathsf{parseUCon}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow c_i :: [\omega]\kappa'_i\}_{0\leq i<n_{\text{con}}}$ — by Lemma C.45 on (2) and (8)

   The conclusions hold as follows:

   (a) (9)
   (b) (10)
   (c) (11)
   (d) (2)
   (e) (3) and (4)
   (f) Choose $e'' = x$ for fresh x. Then $e = [\omega', e/x]x$ for any ω'.
   (g) $\mathsf{fv}(x) = x$ and $x \subset \{x\} \cup \{k_i\}_{0\leq i<n_{\text{kind}}} \cup \{u_i\}_{0\leq i<n_{\text{con}}} \cup \text{dom}(\Omega)$ by definition.

2. There is only one case.

   Case (C.42)**.**

   (1) $\grave{r} = \texttt{prrule}(p.e)$ — by assumption
   (2) $r = p \Rightarrow e$ — by assumption

(3) $\Omega \vdash p : \tau \dashv\!\mid \Omega'$ by assumption

(4) $\Omega \cup \Omega' \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau'$ by assumption

(5) $\mathsf{summary}(\grave{r}) = \mathsf{summary}(e)$ by definition

(6) $\{\hat{\Omega} \vdash \mathsf{parseUKind}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow \kappa_i \ \mathsf{kind}\}_{0 \leq i < n_{\text{kind}}}$ by IH, part 1 on (4) and (5)

(7) $\{\Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}};\hat{\Omega};b} \grave{\kappa}'_i \rightsquigarrow \kappa'_i \ \mathsf{kind}\}_{0 \leq i < n_{\text{con}}}$ by IH, part 1 on (4) and (5)

(8) $\{\hat{\Omega} \vdash \mathsf{parseUCon}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow c_i :: [\omega]\kappa'_i\}_{0 \leq i < n_{\text{con}}}$ by IH, part 1 on (4) and (5)

(9) $\{\Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}};\hat{\Omega};b} \grave{\tau}_i \rightsquigarrow \tau_i :: \mathsf{Type}\}_{0 \leq i < n_{\text{exp}}}$ by IH, part 1 on (4) and (5)

(10) $\{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathsf{parseUExp}(\mathsf{subseq}(b; m''_i; n''_i)) \rightsquigarrow e_i : [\omega]\tau_i\}_{0 \leq i < n_{\text{exp}}}$ by IH, part 1 on (4) and (5)

(11) $e = [\{\kappa_i/k_i\}_{0 \leq i < n_{\text{kind}}}, \{c_i/u_i\}_{0 \leq i < n_{\text{con}}}, \{e_i/x_i\}_{0 \leq i < n_{\text{exp}}}, \omega]e''$ for some e'' and fresh $\{k_i\}_{0 \leq i < n_{\text{kind}}}$ and fresh $\{u_i\}_{0 \leq i < n_{\text{con}}}$ and fresh $\{x_i\}_{0 \leq i < n_{\text{exp}}}$ by IH, part 1 on (4) and (5)

(12) $\mathsf{fv}(e'') \subset \{k_i\}_{0 \leq i < n_{\text{kind}}} \cup \{u_i\}_{0 \leq i < n_{\text{con}}} \cup \{x_i\}_{0 \leq i < n_{\text{exp}}} \cup \mathsf{dom}(\Omega \cup \Omega')$ by IH, part 1 on (4) and (5)

(13) $\mathsf{fv}(e'') \subset \{k_i\}_{0 \leq i < n_{\text{kind}}} \cup \{u_i\}_{0 \leq i < n_{\text{con}}} \cup \{x_i\}_{0 \leq i < n_{\text{exp}}} \cup \mathsf{dom}(\Omega) \cup \mathsf{dom}(\Omega')$ by distributivity of union

(14) $r = [\{\kappa_i/k_i\}_{0 \leq i < n_{\text{kind}}}, \{c_i/u_i\}_{0 \leq i < n_{\text{con}}}, \{e_i/x_i\}_{0 \leq i < n_{\text{exp}}}, \omega]p \Rightarrow e''$ by definition of substitution

(15) $\mathsf{dom}(\Omega') = \mathsf{patvars}(p)$ by Lemma C.5 on (3)

(16) $\mathsf{fv}(p \Rightarrow e'') \subset \{k_i\}_{0 \leq i < n_{\text{kind}}} \cup \{u_i\}_{0 \leq i < n_{\text{con}}} \cup \{x_i\}_{0 \leq i < n_{\text{exp}}} \cup \mathsf{dom}(\Omega)$ by Definition of $\mathsf{fv}(r)$ and (15) and (13)

The conclusions hold as follows:

(a) (6)

(b) (7)

(c) (8)

(d) (9)

(e) (10)

(f) Choose $r' = p \Rightarrow e''$, then by (14)

(g) (16)

$\square$

Theorem C.47 (peTSM Abstract 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**)* $\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{e} \rightsquigarrow \epsilon \; @ \; \texttt{type}(\tau')$ *and* $\Omega_{app} \vdash e : \tau'$ *for* τ' *such that* $\Omega_{app} \vdash \tau' <: \tau$.
4. $\Omega_{app} \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \Downarrow \epsilon_{normal}$
5. $\mathsf{tsmdef}(\epsilon_{normal}) = a$
6. $\Psi = \Psi', a \hookrightarrow \texttt{petsm}(\rho; e_{parse})$
7. $b \downarrow_{\mathsf{Body}} e_{body}$
8. $e_{parse}(e_{body}) \Downarrow \texttt{inj}[\mathtt{SuccessE}](e_{pproto})$
9. $e_{pproto} \uparrow_{\mathsf{PPrExpr}} \grave{e}$
10. $\Omega_{app} \vdash^{\mathsf{Exp}}_{\Psi} \grave{e} \looparrowright_{\epsilon_{normal}} \grave{e} \; ? \; \texttt{type}(\tau_{proto}) \dashv \omega : \Omega_{params}$
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. $\mathsf{summary}(\grave{e}) = \{\texttt{splicedk}[m_i; n_i]\}_{0 \le i < n_{kind}} \cup \{\texttt{splicedc}[m'_i; n'_i; \grave{\kappa}'_i]\}_{0 \le i < n_{con}} \cup \{\texttt{splicede}[m''_i; n''_i; \grave{\tau}_i]\}_{0 \le i < n_{exp}}$
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**)* $\{\Omega_{params} \vdash^{\omega:\Omega_{params};\hat{\Omega};b} \grave{\tau}_i \rightsquigarrow \tau_i :: \mathtt{Type}\}_{0 \le i < n_{exp}}$ *and* $\{\Omega_{app} \vdash [\omega]\tau_i :: \mathtt{Type}\}_{0 \le i < n_{exp}}$
20. *(**Typing 2**)* $\{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathsf{parseUExp}(\mathsf{subseq}(b; m''_i; n''_i)) \rightsquigarrow e_i : [\omega]\tau_i\}_{0 \le i < n_{exp}}$ *and* $\{\Omega_{app} \vdash e_i : [\omega]\tau_i\}_{0 \le i < n_{exp}}$
21. *(**Capture Avoidance**)* $e = [\{\kappa_i/k_i\}_{0 \le i < n_{kind}}, \{c_i/u_i\}_{0 \le i < n_{con}}, \{e_i/x_i\}_{0 \le i < n_{exp}}, \omega]e''$ *for some* e'' *and fresh* $\{k_i\}_{0 \le i < n_{kind}}$ *and fresh* $\{u_i\}_{0 \le i < n_{con}}$ *and fresh* $\{x_i\}_{0 \le i < n_{exp}}$
22. *(**Context Independence**)*

$$\mathsf{fv}(e'') \subset \{k_i\}_{0 \le i < n_{kind}} \cup \{u_i\}_{0 \le i < n_{con}} \cup \{x_i\}_{0 \le i < n_{exp}} \cup dom(\Omega_{params})$$

Proof. By rule induction over Rules (C.19). There are two rules that apply.

Case (C.19a).

(1) $\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}$ 'b' $\rightsquigarrow e : \tau'$	by assumption
(2) $\Omega \vdash \tau' <: \tau$	by assumption
(3) $\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\mathsf{app}} \rangle$	by IH on (1)
(4) $\hat{\Psi} = \langle \mathcal{A}; \Psi \rangle$	by IH on (1)

(5) $\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{e} \leadsto \epsilon \mathbin{@} \mathtt{type}(\tau'')$ and $\Omega_{\text{app}} \vdash e : \tau''$ for τ'' such that $\Omega_{\text{app}} \vdash \tau'' <: \tau'$. by IH on (1)

(6) $\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \Downarrow \epsilon_{\text{normal}}$ by IH on (1)

(7) $\mathsf{tsmdef}(\epsilon_{\text{normal}}) = a$ by IH on (1)

(8) $\Psi = \Psi', a \hookrightarrow \mathtt{petsm}(\rho; e_{\text{parse}})$ by IH on (1)

(9) $b \downarrow_{\mathsf{Body}} e_{\text{body}}$ by IH on (1)

(10) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \mathtt{inj}[\mathtt{SuccessE}](e_{\text{pproto}})$ by IH on (1)

(11) $e_{\text{pproto}} \uparrow_{\mathsf{PPrExpr}} \dot{e}$ by IH on (1)

(12) $\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \dot{e} \looparrowright_{\epsilon_{\text{normal}}} \grave{e} \mathbin{?} \mathtt{type}(\tau_{\text{proto}}) \dashv \omega : \Omega_{\text{params}}$ by IH on (1)

(13) $\mathsf{seg}(\grave{e})$ segments b by IH on (1)

(14) $\Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}};\hat{\Omega};\hat{\Psi};\hat{\Phi};b} \grave{e} \leadsto e' : \tau_{\text{proto}}$ by IH on (1)

(15) $e = [\omega]e'$ by IH on (1)

(16) $\tau' = [\omega]\tau_{\text{proto}}$ by IH on (1)

(17) $\mathsf{summary}(\grave{e}) = \{\mathtt{splicedk}[m_i; n_i]\}_{0 \le i < n_{\text{kind}}} \cup \{\mathtt{splicedc}[m'_i; n'_i; \grave{\kappa}'_i]\}_{0 \le i < n_{\text{con}}} \cup \{\mathtt{splicede}[m''_i; n''_i; \grave{\tau}_i]\}_{0 \le i < n_{\text{exp}}}$ by IH on (1)

(18) $\{\hat{\Omega} \vdash \mathsf{parseUKind}(\mathsf{subseq}(b; m_i; n_i)) \leadsto \kappa_i \ \mathsf{kind}\}_{0 \le i < n_{\text{kind}}}$ and $\{\Omega_{\text{app}} \vdash \kappa_i \ \mathsf{kind}\}_{0 \le i < n_{\text{kind}}}$ by IH on (1)

(19) $\{\Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}};\hat{\Omega};b} \grave{\kappa}'_i \leadsto \kappa'_i \ \mathsf{kind}\}_{0 \le i < n_{\text{con}}}$ and $\{\Omega_{\text{app}} \vdash [\omega]\kappa'_i \ \mathsf{kind}\}_{0 \le i < n_{\text{con}}}$ by IH on (1)

(20) $\{\hat{\Omega} \vdash \mathsf{parseUCon}(\mathsf{subseq}(b; m'_i; n'_i)) \leadsto c_i :: [\omega]\kappa'_i\}_{0 \le i < n_{\text{con}}}$ and $\{\Omega_{\text{app}} \vdash c_i :: [\omega]\kappa'_i\}_{0 \le i < n_{\text{con}}}$ by IH on (1)

(21) $\{\Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}};\hat{\Omega};b} \grave{\tau}_i \leadsto \tau_i :: \mathtt{Type}\}_{0 \le i < n_{\text{exp}}}$ and $\{\Omega_{\text{app}} \vdash [\omega]\tau_i :: \mathtt{Type}\}_{0 \le i < n_{\text{exp}}}$ by IH on (1)

(22) $\{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathsf{parseUExp}(\mathsf{subseq}(b; m''_i; n''_i)) \leadsto e_i : [\omega]\tau_i\}_{0 \le i < n_{\text{exp}}}$ and $\{\Omega_{\text{app}} \vdash e_i : [\omega]\tau_i\}_{0 \le i < n_{\text{exp}}}$ by IH on (1)

(23) $e = [\{\kappa_i/k_i\}_{0 \le i < n_{\text{kind}}}, \{c_i/u_i\}_{0 \le i < n_{\text{con}}}, \{e_i/x_i\}_{0 \le i < n_{\text{exp}}}, \omega]e''$ for some e'' and fresh $\{k_i\}_{0 \le i < n_{\text{kind}}}$ and fresh $\{u_i\}_{0 \le i < n_{\text{con}}}$ and fresh $\{x_i\}_{0 \le i < n_{\text{exp}}}$ by IH on (1)

(24) $\mathsf{fv}(e'') \subset \{k_i\}_{0 \le i < n_{\text{kind}}} \cup \{u_i\}_{0 \le i < n_{\text{con}}} \cup \{x_i\}_{0 \le i < n_{\text{exp}}} \cup \mathrm{dom}(\Omega_{\text{params}})$ by IH on (1)

(25) $\Omega_{\text{app}} \vdash \tau'' <: \tau$ by Rule (C.11b) on (2) and (5)

The conclusions hold as follows:

1. (3)
2. (4)
3. Choosing τ'', by (5) and (25)
4. (6)
5. (7)
6. (8)
7. (9)

8. (10)
9. (11)
10. (12)
11. (13)
12. (14)
13. (15)
14. (16)
15. (17)
16. (18)
17. (19)
18. (20)
19. (21)
20. (22)
21. (23)
22. (24)

Case (C.19p).

(1) $e = [\omega]e'$	by assumption
(2) $\tau = [\omega]\tau_{\text{proto}}$	by assumption
(3) $\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\text{app}} \rangle$	by assumption
(4) $\hat{\Psi} = \langle \mathcal{A}; \Psi \rangle$	by assumption
(5) $\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{e} \rightsquigarrow \epsilon \mathbin{@} \mathtt{type}(\tau_{\text{final}})$	by assumption
(6) $\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \Downarrow \epsilon_{\text{normal}}$	by assumption
(7) $\mathsf{tsmdef}(\epsilon_{\text{normal}}) = a$	by assumption
(8) $\Psi = \Psi', a \hookrightarrow \mathtt{petsm}(\rho; e_{\text{parse}})$	by assumption
(9) $b \downarrow_{\mathsf{Body}} e_{\text{body}}$	by assumption
(10) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \mathtt{inj}[\mathtt{SuccessE}](e_{\text{pproto}})$	by assumption
(11) $e_{\text{pproto}} \uparrow_{\mathsf{PPrExpr}} \dot{e}$	by assumption
(12) $\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \dot{e} \looparrowright_{\epsilon_{\text{normal}}} \grave{e} \mathbin{?} \mathtt{type}(\tau_{\text{proto}}) \dashv \omega : \Omega_{\text{params}}$	by assumption
(13) $\mathsf{seg}(\grave{e})$ segments b	by assumption
(14) $\Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}}; \hat{\Omega}; \hat{\Psi}; \hat{\Phi}; b} \grave{e} \rightsquigarrow e' : \tau_{\text{proto}}$	by assumption
(15) $\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} \mathtt{type}(\tau_{\text{final}})$	by Theorem C.38 on (5)
(16) $\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \epsilon_{\text{normal}} \mathbin{@} \mathtt{type}(\tau_{\text{final}})$	by Corollary C.29 on (15) and (6)
(17) $\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \epsilon_{\text{normal}} \mathbin{@} [\omega]\mathtt{type}(\tau_{\text{proto}})$	by Lemma C.42 on (12)
(18) $\mathtt{type}(\tau_{\text{final}}) = [\omega]\mathtt{type}(\tau_{\text{proto}})$	by Theorem C.25 on (16) and (17)

(19) $\tau_{\text{final}} = [\omega]\tau_{\text{proto}} = \tau$ by definition of substitution and (18) and (2)

(20) $\Omega_{\text{app}} \vdash \texttt{type}(\tau_{\text{final}})\ \mathsf{tsmty}$ by Lemma C.23 on (15)

(21) $\Omega_{\text{app}} \vdash \tau :: \texttt{Type}$ by Inversion of Rule (C.25a) on (20)

(22) $\Omega_{\text{app}} \vdash \tau \equiv \tau :: \texttt{Type}$ by Rule (C.10a) on (21)

(23) $\Omega_{\text{app}} \vdash \tau <: \tau$ by Rule (C.11a) on (22)

(24) $\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}\ \text{`}b\text{`} \leadsto e : \tau$ by assumption

(25) $\Omega_{\text{app}} \vdash e : \tau$ by Theorem C.43 on (24)

(26) $\Omega_{\text{app}} \vdash \omega : \Omega_{\text{params}}$ by Lemma C.42 on (12)

(27) $\mathsf{summary}(\grave{e}) = \{\texttt{splicedk}[m_i; n_i]\}_{0 \le i < n_{\text{kind}}} \cup \{\texttt{splicedc}[m'_i; n'_i; \grave{\kappa}'_i]\}_{0 \le i < n_{\text{con}}} \cup \{\texttt{splicede}[m''_i; n''_i; \grave{\tau}_i]\}_{0 \le i < n_{\text{exp}}}$ by definition

(28) $\{\hat{\Omega} \vdash \mathsf{parseUKind}(\mathsf{subseq}(b; m_i; n_i)) \leadsto \kappa_i\ \mathsf{kind}\}_{0 \le i < n_{\text{kind}}}$ by Lemma C.46 on (14) and (27)

(29) $\{\Omega_{\text{app}} \vdash \kappa_i\ \mathsf{kind}\}_{0 \le i < n_{\text{kind}}}$ by Theorem C.35 over (28)

(30) $\{\Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}};\hat{\Omega};b} \grave{\kappa}'_i \leadsto \kappa'_i\ \mathsf{kind}\}_{0 \le i < n_{\text{con}}}$ by Lemma C.46 on (14) and (27)

(31) $\{\Omega_{\text{app}} \cup \Omega_{\text{params}} \vdash \kappa'_i\ \mathsf{kind}\}_{0 \le i < n_{\text{con}}}$ by Theorem C.35 over (30)

(32) $\{\Omega_{\text{app}} \vdash [\omega]\kappa'_i\ \mathsf{kind}\}_{0 \le i < n_{\text{con}}}$ by Lemma C.3 over (31) and (26)

(33) $\{\hat{\Omega} \vdash \mathsf{parseUCon}(\mathsf{subseq}(b; m'_i; n'_i)) \leadsto c_i :: [\omega]\kappa'_i\}_{0 \le i < n_{\text{con}}}$ by Lemma C.46 on (14) and (27)

(34) $\{\Omega_{\text{app}} \vdash c_i :: [\omega]\kappa'_i\}_{0 \le i < n_{\text{con}}}$ by Theorem C.35 over (33)

(35) $\{\Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}};\hat{\Omega};b} \grave{\tau}_i \leadsto \tau_i :: \texttt{Type}\}_{0 \le i < n_{\text{exp}}}$ by Lemma C.46 on (14) and (27)

(36) $\{\Omega_{\text{app}} \cup \Omega_{\text{params}} \vdash \tau_i :: \texttt{Type}\}_{0 \le i < n_{\text{exp}}}$ by Theorem C.35 over (35)

(37) $\{\Omega_{\text{app}} \vdash [\omega]\tau_i :: \texttt{Type}\}_{0 \le i < n_{\text{exp}}}$ by Lemma C.3 over (36) and (26)

(38) $\{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathsf{parseUExp}(\mathsf{subseq}(b; m''_i; n''_i)) \leadsto e_i : [\omega]\tau_i\}_{0 \le i < n_{\text{exp}}}$ by Lemma C.46 on (14) and (27)

(39) $\{\Omega_{\mathsf{app}} \vdash e_i : [\omega]\tau_i\}_{0 \leq i < n_{\mathsf{exp}}}$ by Theorem C.43 over (38)

(40) $e = [\{\kappa_i/k_i\}_{0 \leq i < n_{\mathsf{kind}}}, \{c_i/u_i\}_{0 \leq i < n_{\mathsf{con}}}, \{e_i/x_i\}_{0 \leq i < n_{\mathsf{exp}}}, \omega]e''$ for some e'' and fresh $\{k_i\}_{0 \leq i < n_{\mathsf{kind}}}$ and fresh $\{u_i\}_{0 \leq i < n_{\mathsf{con}}}$ and fresh $\{x_i\}_{0 \leq i < n_{\mathsf{exp}}}$ by Lemma C.46 on (14) and (27)

(41) $\mathsf{fv}(e'') \subset \{k_i\}_{0 \leq i < n_{\mathsf{kind}}} \cup \{u_i\}_{0 \leq i < n_{\mathsf{con}}} \cup \{x_i\}_{0 \leq i < n_{\mathsf{exp}}} \cup \mathrm{dom}(\Omega_{\mathsf{params}})$ by Lemma C.46 on (14) and (27)

The conclusions hold as follows:

1. (3)
2. (4)
3. Choosing τ', by (5), (19), (25) and (23)
4. (6)
5. (7)
6. (8)
7. (9)
8. (10)
9. (11)
10. (12)
11. (13)
12. (14)
13. (1)
14. (2)
15. (27)
16. (28) and (29)
17. (30) and (32)
18. (33) and (34)
19. (35) and (37)
20. (38) and (39)
21. (40)
22. (41)

$\square$

Lemma C.48 (Proto-Pattern Decomposition). *If* $\grave{p} \rightsquigarrow p : \tau \dashv\!|^{\omega:\Omega_{params};\hat{\Omega};\hat{\Phi};b} \hat{\Omega}'$ *where*

$$\begin{array}{rcl} \mathsf{summary}(\grave{p}) & = & \{\mathtt{splicedk}[m_i; n_i]\}_{0 \leq i < n_{kind}} \\ & \cup & \{\mathtt{splicedc}[m'_i; n'_i; \grave{\kappa}'_i]\}_{0 \leq i < n_{con}} \\ & \cup & \{\mathtt{splicedp}[m''_i; n''_i; \grave{\tau}_i]\}_{0 \leq i < n_{pat}} \end{array}$$

then

1. $\{\hat{\Omega} \vdash \mathsf{parseUKind}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow \kappa_i \;\mathsf{kind}\}_{0 \leq i < n_{kind}}$
2. $\{\Omega_{params} \vdash^{\omega:\Omega_{params};\hat{\Omega};b} \grave{\kappa}'_i \rightsquigarrow \kappa'_i \;\mathsf{kind}\}_{0 \leq i < n_{con}}$

3. $\{\hat{\Omega} \vdash \mathsf{parseUCon}(\mathsf{subseq}(b; m_i'; n_i')) \rightsquigarrow c_i :: [\omega]\kappa_i'\}_{0 \le i < n_{con}}$
4. $\{\Omega_{params} \vdash^{\omega:\Omega_{params};\hat{\Omega};b} \grave{\tau}_i \rightsquigarrow \tau_i :: \mathtt{Type}\}_{0 \le i < n_{pat}}$
5. $\{\hat{\Omega} \vdash_{\hat{\Phi}} \mathsf{parseUPat}(\mathsf{subseq}(b; m_i''; n_i'')) \rightsquigarrow p_i : [\omega]\tau_i \dashv\vdash \langle \emptyset; \emptyset; \mathcal{G}_i; \Omega_i \rangle\}_{0 \le i < n_{pat}}$
6. $\hat{\Omega}' = \langle \emptyset; \emptyset; \biguplus_{0 \le i < n_{pat}} \mathcal{G}_i; \bigcup_{0 \le i < n_{pat}} \Omega_i \rangle$

Proof. By rule induction over Rules (C.43).

Case (C.43a) **through** (C.43d)**.** In each of these cases, we apply the IH to or over each premise and then gather the sets of conclusions, applying the identification convention as necessary.

Case (C.43e)**.**

(1)	$\grave{p} = \mathtt{splicedp}[m; n; \grave{\tau}]$	by assumption
(2)	$\Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}};\hat{\Omega};b} \grave{\tau} \rightsquigarrow \tau :: \mathtt{Type}$	by assumption
(3)	$\mathsf{parseUPat}(\mathsf{subseq}(b; m; n)) = \hat{p}$	by assumption
(4)	$\hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : [\omega]\tau \dashv\vdash \hat{\Omega}'$	by assumption
(5)	$\mathsf{summary}(\grave{p}) = \mathsf{summary}(\grave{\tau}) \cup \{\mathtt{splicedp}[m; n; \grave{\tau}]\}$	by definition
(6)	$\mathsf{summary}(\grave{\tau}) = \{\mathtt{splicedk}[m_i; n_i]\}_{0 \le i < n_{\text{kind}}} \cup \{\mathtt{splicedc}[m_i'; n_i'; \grave{\kappa}_i']\}_{0 \le i < n_{\text{con}}}$	by definition
(7)	$\{\hat{\Omega} \vdash \mathsf{parseUKind}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow \kappa_i \ \mathsf{kind}\}_{0 \le i < n_{\text{kind}}}$	by Lemma C.45 on (2) and (6)
(8)	$\{\Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}};\hat{\Omega};b} \grave{\kappa}_i' \rightsquigarrow \kappa_i' \ \mathsf{kind}\}_{0 \le i < n_{\text{con}}}$	by Lemma C.45 on (2) and (6)
(9)	$\{\hat{\Omega} \vdash \mathsf{parseUCon}(\mathsf{subseq}(b; m_i'; n_i')) \rightsquigarrow c_i :: [\omega]\kappa_i'\}_{0 \le i < n_{\text{con}}}$	by Lemma C.45 on (2) and (6)

The conclusions hold as follows:

1. (7)
2. (8)
3. (9)
4. (2)
5. (3) and (4)
6. (4) because $n_{\text{pat}} = 1$

□

Theorem C.49 (ppTSM Abstract Reasoning Principles)**.** *If* $\hat{\Omega} \vdash_{\hat{\Phi}} \hat{e}$ *'b'* $\rightsquigarrow p : \tau \dashv\vdash \hat{\Omega}'$ *then:*

1. $\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{app} \rangle$
2. $\hat{\Phi} = \langle \mathcal{A}; \Phi \rangle$
3. *(**Typing 1**)* $\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \hat{e} \rightsquigarrow \epsilon$ @ $\mathtt{type}(\tau')$ *and* $\Omega_{app} \vdash p : \tau' \dashv\vdash \hat{\Omega}'$ *for* τ' *such that* $\Omega_{app} \vdash \tau' <: \tau$
4. $\Omega_{app} \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \Downarrow \epsilon_{normal}$
5. $\mathsf{tsmdef}(\epsilon_{normal}) = a$
6. $\Phi = \Phi', a \hookrightarrow \mathtt{pptsm}(\rho; e_{parse})$

7. $b \downarrow_{\mathsf{Body}} e_{body}$
8. $e_{parse}(e_{body}) \Downarrow \mathtt{inj}[\mathtt{SuccessP}](e_{pproto})$
9. $e_{pproto} \uparrow_{\mathsf{PPrPat}} \grave{p}$
10. $\Omega_{app} \vdash^{\mathsf{Pat}}_{\Phi} \grave{p} \hookrightarrow_{\epsilon_{normal}} \grave{p} \mathrel{?} \mathtt{type}(\tau_{proto}) \dashv \omega : \Omega_{params}$
11. *(Segmentation)* $\mathsf{seg}(\grave{p})$ segments b
12. $\grave{p} \leadsto p : \tau_{proto} \dashv\mid^{\omega:\Omega_{params};\hat{\Omega};\hat{\Phi};b} \hat{\Omega}'$
13. $\tau' = [\omega]\tau_{proto}$
14. $\mathsf{summary}(\grave{p}) = \{\mathtt{splicedk}[m_i; n_i]\}_{0 \le i < n_{kind}} \cup \{\mathtt{splicedc}[m'_i; n'_i; \grave{\kappa}'_i]\}_{0 \le i < n_{con}} \cup \{\mathtt{splicedp}[m''_i; n''_i; \grave{\tau}_i]\}_{0 \le i < n_{pat}}$
15. *(Kinding 1)* $\{\hat{\Omega} \vdash \mathsf{parseUKind}(\mathsf{subseq}(b; m_i; n_i)) \leadsto \kappa_i \;\mathsf{kind}\}_{0 \le i < n_{kind}}$ *and* $\{\Omega_{app} \vdash \kappa_i \;\mathsf{kind}\}_{0 \le i < n_{kind}}$
16. *(Kinding 2)* $\{\Omega_{params} \vdash^{\omega:\Omega_{params};\hat{\Omega};b} \grave{\kappa}'_i \leadsto \kappa'_i \;\mathsf{kind}\}_{0 \le i < n_{con}}$ *and* $\{\Omega_{app} \vdash [\omega]\kappa'_i \;\mathsf{kind}\}_{0 \le i < n_{con}}$
17. *(Kinding 3)* $\{\hat{\Omega} \vdash \mathsf{parseUCon}(\mathsf{subseq}(b; m'_i; n'_i)) \leadsto c_i :: [\omega]\kappa'_i\}_{0 \le i < n_{con}}$ *and* $\{\Omega_{app} \vdash c_i :: [\omega]\kappa'_i\}_{0 \le i < n_{con}}$
18. *(Kinding 4)* $\{\Omega_{params} \vdash^{\omega:\Omega_{params};\hat{\Omega};b} \grave{\tau}_i \leadsto \tau_i :: \mathit{Type}\}_{0 \le i < n_{pat}}$ *and* $\{\Omega_{app} \vdash [\omega]\tau_i :: \mathit{Type}\}_{0 \le i < n_{pat}}$
19. *(Typing 2)* $\{\hat{\Omega} \vdash_{\hat{\Phi}} \mathsf{parseUPat}(\mathsf{subseq}(b; m''_i; n''_i)) \leadsto p_i : [\omega]\tau_i \dashv\mid \langle \emptyset; \emptyset; \mathcal{G}_i; \Omega_i \rangle\}_{0 \le i < n_{pat}}$ *and* $\{\Omega_{app} \vdash p_i : [\omega]\tau_i \dashv\mid \Omega_i\}_{0 \le i < n_{pat}}$
20. *(No Hidden Bindings)* $\hat{\Omega}' = \langle \emptyset; \emptyset; \biguplus_{0 \le i < n_{pat}} \mathcal{G}_i; \bigcup_{0 \le i < n_{pat}} \Omega_i \rangle$

Proof. By rule induction over Rules (C.21). There are two rules that apply.

Case (C.21a).

(1) $\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega \rangle$	by assumption
(2) $\hat{\Omega} \vdash_{\hat{\Phi}} \hat{e}$ ‘b‘ $\leadsto p : \tau' \dashv\mid \hat{\Omega}'$	by assumption
(3) $\Omega \vdash \tau' <: \tau$	by assumption
(4) $\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\text{app}} \rangle$	by IH on (2)
(5) $\hat{\Phi} = \langle \mathcal{A}; \Phi \rangle$	by IH on (2)
(6) $\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \hat{e} \leadsto \epsilon \mathbin{@} \mathtt{type}(\tau'')$ and $\Omega_{\text{app}} \vdash p : \tau'' \dashv\mid \hat{\Omega}'$ for τ'' such that $\Omega_{\text{app}} \vdash \tau'' <: \tau'$	by IH on (2)
(7) $\Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \Downarrow \epsilon_{\text{normal}}$	by IH on (2)
(8) $\mathsf{tsmdef}(\epsilon_{\text{normal}}) = a$	by IH on (2)
(9) $\Phi = \Phi', a \hookrightarrow \mathtt{pptsm}(\rho; e_{\text{parse}})$	by IH on (2)
(10) $b \downarrow_{\mathsf{Body}} e_{\text{body}}$	by IH on (2)
(11) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \mathtt{inj}[\mathtt{SuccessP}](e_{\text{pproto}})$	by IH on (2)
(12) $e_{\text{pproto}} \uparrow_{\mathsf{PPrPat}} \grave{p}$	by IH on (2)
(13) $\Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi} \grave{p} \hookrightarrow_{\epsilon_{\text{normal}}} \grave{p} \mathrel{?} \mathtt{type}(\tau_{\text{proto}}) \dashv \omega : \Omega_{\text{params}}$	by IH on (2)
(14) $\mathsf{seg}(\grave{p})$ segments b	by IH on (2)
(15) $\grave{p} \leadsto p : \tau_{\text{proto}} \dashv\mid^{\omega:\Omega_{\text{params}};\hat{\Omega};\hat{\Phi};b} \hat{\Omega}'$	by IH on (2)
(16) $\tau' = [\omega]\tau_{\text{proto}}$	by IH on (2)
(17) $\mathsf{summary}(\grave{p}) = \{\mathtt{splicedk}[m_i; n_i]\}_{0 \le i < n_{\text{kind}}} \cup$	

$\{\mathtt{splicedc}[m'_i; n'_i; \grave{\kappa}'_i]\}_{0 \leq i < n_{\text{con}}} \cup \{\mathtt{splicedp}[m''_i; n''_i; \grave{\tau}_i]\}_{0 \leq i < n_{\text{pat}}}$ by IH on (2)

(18) $\{\hat{\Omega} \vdash \mathsf{parseUKind}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow \kappa_i \ \mathsf{kind}\}_{0 \leq i < n_{\text{kind}}}$ and $\{\Omega_{\text{app}} \vdash \kappa_i \ \mathsf{kind}\}_{0 \leq i < n_{\text{kind}}}$ by IH on (2)

(19) $\{\Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}}; \hat{\Omega}; b} \grave{\kappa}'_i \rightsquigarrow \kappa'_i \ \mathsf{kind}\}_{0 \leq i < n_{\text{con}}}$ and $\{\Omega_{\text{app}} \vdash [\omega]\kappa'_i \ \mathsf{kind}\}_{0 \leq i < n_{\text{con}}}$ by IH on (2)

(20) $\{\hat{\Omega} \vdash \mathsf{parseUCon}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow c_i :: [\omega]\kappa'_i\}_{0 \leq i < n_{\text{con}}}$ and $\{\Omega_{\text{app}} \vdash c_i :: [\omega]\kappa'_i\}_{0 \leq i < n_{\text{con}}}$ by IH on (2)

(21) $\{\Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}}; \hat{\Omega}; b} \grave{\tau}_i \rightsquigarrow \tau_i :: \mathtt{Type}\}_{0 \leq i < n_{\text{pat}}}$ and $\{\Omega_{\text{app}} \vdash [\omega]\tau_i :: \mathtt{Type}\}_{0 \leq i < n_{\text{pat}}}$ by IH on (2)

(22) $\{\hat{\Omega} \vdash_{\hat{\Phi}} \mathsf{parseUPat}(\mathsf{subseq}(b; m''_i; n''_i)) \rightsquigarrow p_i : [\omega]\tau_i \dashv\!\vert \langle \emptyset; \emptyset; \mathcal{G}_i; \Omega_i \rangle\}_{0 \leq i < n_{\text{pat}}}$ and $\{\Omega_{\text{app}} \vdash p_i : [\omega]\tau_i \dashv\!\vert \Omega_i\}_{0 \leq i < n_{\text{pat}}}$ by IH on (2)

(23) $\hat{\Omega}' = \langle \emptyset; \emptyset; \uplus_{0 \leq i < n_{\text{pat}}} \mathcal{G}_i; \bigcup_{0 \leq i < n_{\text{pat}}} \Omega_i \rangle$ by IH on (2)

(24) $\Omega_{\text{app}} \vdash \tau'' <: \tau$ by Rule (C.11b) on (6) and (3)

The conclusions hold as follows:

1. (4)
2. (5)
3. Choosing τ'', by (6) and (24)
4. (7)
5. (8)
6. (9)
7. (10)
8. (11)
9. (12)
10. (13)
11. (14)
12. (15)
13. (16)
14. (17)
15. (18)
16. (19)
17. (20)
18. (21)
19. (22)
20. (23)

Case (C.21g).

(1) $\hat{\Omega} = \langle \mathcal{M}; \mathcal{D}; \mathcal{G}; \Omega_{\text{app}} \rangle$ by assumption

(2) $\hat{\Phi} = \langle \mathcal{A}; \Phi \rangle$ by assumption

(3) $\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \hat{e} \rightsquigarrow e \ @ \ \mathtt{type}(\tau_{\text{final}})$ by assumption

(4) $\Omega_{\text{app}} \vdash^{\text{Pat}}_{\Phi} \epsilon \Downarrow \epsilon_{\text{normal}}$ by assumption

(5) $\text{tsmdef}(\epsilon_{\text{normal}}) = a$ by assumption

(6) $\Phi = \Phi', a \hookrightarrow \texttt{pptsm}(\rho; e_{\text{parse}})$ by assumption

(7) $b \downarrow_{\text{Body}} e_{\text{body}}$ by assumption

(8) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}[\texttt{SuccessP}](e_{\text{pproto}})$ by assumption

(9) $e_{\text{pproto}} \uparrow_{\text{PPrPat}} \dot{p}$ by assumption

(10) $\Omega_{\text{app}} \vdash^{\text{Pat}}_{\Phi} \dot{p} \hookrightarrow_{\epsilon_{\text{normal}}} \grave{p} \mathrel{?} \texttt{type}(\tau_{\text{proto}}) \dashv \omega : \Omega_{\text{params}}$ by assumption

(11) $\text{seg}(\grave{p})$ segments b by assumption

(12) $\grave{p} \rightsquigarrow p : \tau_{\text{proto}} \dashv\mid^{\omega:\Omega_{\text{params}};\hat{\Omega};\hat{\Phi};b} \hat{\Omega}'$ by assumption

(13) $\tau = [\omega]\tau_{\text{proto}}$ by assumption

(14) $\Omega_{\text{app}} \vdash^{\text{Pat}}_{\Psi} \epsilon \mathrel{@} \texttt{type}(\tau_{\text{final}})$ by Theorem C.39 on (3)

(15) $\Omega_{\text{app}} \vdash^{\text{Pat}}_{\Psi} \epsilon_{\text{normal}} \mathrel{@} \texttt{type}(\tau_{\text{final}})$ by Corollary C.32 on (14) and (4)

(16) $\Omega_{\text{app}} \vdash^{\text{Pat}}_{\Psi} \epsilon_{\text{normal}} \mathrel{@} [\omega]\texttt{type}(\tau_{\text{proto}})$ by Lemma C.40 on (10)

(17) $\texttt{type}(\tau_{\text{final}}) = [\omega]\texttt{type}(\tau_{\text{proto}})$ by Theorem C.26 on (15) and (16)

(18) $\tau_{\text{final}} = [\omega]\tau_{\text{proto}} = \tau$ by definition of substitution and (17) and (13)

(19) $\Omega_{\text{app}} \vdash \texttt{type}(\tau_{\text{final}})$ tsmty by Lemma C.24 on (14)

(20) $\Omega_{\text{app}} \vdash \tau :: \texttt{Type}$ by Inversion of Rule (C.25a) on (19)

(21) $\Omega_{\text{app}} \vdash \tau \equiv \tau :: \texttt{Type}$ by Rule (C.10a) on (20)

(22) $\Omega_{\text{app}} \vdash \tau <: \tau$ by Rule (C.11a) on (21)

(23) $\hat{\Omega} \vdash_{\hat{\Phi}} \hat{\epsilon}\ \text{`}b\text{`} \rightsquigarrow p : \tau \dashv\mid \hat{\Omega}'$ by assumption

(24) $\Omega_{\text{app}} \vdash p : \tau \dashv\mid \hat{\Omega}'$ by Theorem C.41 on (23)

(25) $\Omega_{\text{app}} \vdash \omega : \Omega_{\text{params}}$ by Lemma C.40 on (10)

(26) $\text{summary}(\grave{p}) = \{\texttt{splicedk}[m_i; n_i]\}_{0 \leq i < n_{\text{kind}}} \cup \{\texttt{splicedc}[m'_i; n'_i; \grave{\kappa}'_i]\}_{0 \leq i < n_{\text{con}}} \cup \{\texttt{splicedp}[m''_i; n''_i; \grave{\tau}_i]\}_{0 \leq i < n_{\text{pat}}}$ by definition

(27) $\{\hat{\Omega} \vdash \text{parseUKind}(\text{subseq}(b; m_i; n_i)) \rightsquigarrow \kappa_i \text{ kind}\}_{0 \leq i < n_{\text{kind}}}$ by Lemma C.48 on (12) and (26)

(28) $\{\Omega_{\text{app}} \vdash \kappa_i \text{ kind}\}_{0 \leq i < n_{\text{kind}}}$ by Lemma C.35 over (27)

(29) $\{\Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}};\hat{\Omega};b} \grave{\kappa}'_i \leadsto \kappa'_i\ \mathsf{kind}\}_{0\leq i<n_{\text{con}}}$ by Lemma C.48 on (12) and (26)

(30) $\{\Omega_{\text{app}} \cup \Omega_{\text{params}} \vdash \kappa'_i\ \mathsf{kind}\}_{0\leq i<n_{\text{con}}}$ by Theorem C.35 over (29)

(31) $\{\Omega_{\text{app}} \vdash [\omega]\kappa'_i\ \mathsf{kind}\}_{0\leq i<n_{\text{con}}}$ by Lemma C.3 over (30) and (25)

(32) $\{\hat{\Omega} \vdash \mathsf{parseUCon}(\mathsf{subseq}(b;m'_i;n'_i)) \leadsto c_i :: [\omega]\kappa'_i\}_{0\leq i<n_{\text{con}}}$ by Lemma C.48 on (12) and (26)

(33) $\{\Omega_{\text{app}} \vdash [\omega]\tau_i :: \mathtt{Type}\}_{0\leq i<n_{\text{pat}}}$ by Theorem C.35 over (32)

(34) $\{\Omega_{\text{params}} \vdash^{\omega:\Omega_{\text{params}};\hat{\Omega};b} \grave{\tau}_i \leadsto \tau_i :: \mathtt{Type}\}_{0\leq i<n_{\text{pat}}}$ by Lemma C.48 on (12) and (26)

(35) $\{\Omega_{\text{app}} \cup \Omega_{\text{params}} \vdash \tau_i :: \mathtt{Type}\}_{0\leq i<n_{\text{pat}}}$ by Theorem C.35 over (34)

(36) $\{\Omega_{\text{app}} \vdash [\omega]\tau_i :: \mathtt{Type}\}_{0\leq i<n_{\text{pat}}}$ by Lemma C.3 over (35) and (25)

(37) $\{\hat{\Omega} \vdash_{\hat{\Phi}} \mathsf{parseUPat}(\mathsf{subseq}(b;m''_i;n''_i)) \leadsto p_i : [\omega]\tau_i \dashv\!\mid \langle\emptyset;\emptyset;\mathcal{G}_i;\Omega_i\rangle\}_{0\leq i<n_{\text{pat}}}$ by Lemma C.48 on (12) and (26)

(38) $\{\Omega_{\text{app}} \vdash p_i : [\omega]\tau_i \dashv\!\mid \Omega_i\}_{0\leq i<n_{\text{pat}}}$ by Theorem C.41 over (37)

(39) $\hat{\Omega}' = \langle\emptyset;\emptyset;\uplus_{0\leq i<n_{\text{pat}}}\mathcal{G}_i;\bigcup_{0\leq i<n_{\text{pat}}}\Omega_i\rangle$ by Lemma C.48 on (12) and (26)

The conclusions hold as follows:

1. (1)
2. (2)
3. Choosing τ, by (3) and (24) and (22)
4. (4)
5. (5)
6. (6)
7. (7)
8. (8)
9. (9)
10. (10)
11. (11)
12. (12)
13. (13)
14. (26)
15. (27) and (28)
16. (29) and (31)
17. (32) and (33)

18. (34) and (36)
19. (37) and (38)
20. (39)

$\square$

Appendix D

Bidirectional $\mathsf{miniVerse}_{\mathsf{S}}$

D.1 Expanded Language (XL)

The Bidirectional $\mathsf{miniVerse}_{\mathsf{S}}$ expanded language (XL) is the same as the $\mathsf{miniVerse}_{\mathsf{S}}$ XL, which was detailed in Appendix B.1.

D.2 Unexpanded Language (UL)

D.2.1 Syntax

Stylized Syntax

The stylized syntax extends the stylized syntax of the $\mathsf{miniVerse}_{\mathsf{S}}$ UL given in Sec. B.2.1.

Sort			**Stylized Form**	**Description**
UTyp	$\hat{\tau}$	$::=$	$\cdots$	(as in $\mathsf{miniVerse}_{\mathsf{S}}$)
UExp	$\hat{e}$	$::=$	$\cdots$	(as in $\mathsf{miniVerse}_{\mathsf{S}}$)
			$\texttt{implicit syntax}~\hat{a}~\texttt{for expressions in}~\hat{e}$	seTSM designation
			$\texttt{implicit syntax}~\hat{a}~\texttt{for patterns in}~\hat{e}$	spTSM designation
			$/b/$	seTSM unadorned literal
URule	$\hat{r}$	$::=$	$\cdots$	(as in $\mathsf{miniVerse}_{\mathsf{S}}$)
UPat	$\hat{p}$	$::=$	$\cdots$	(as in $\mathsf{miniVerse}_{\mathsf{S}}$)
			$/b/$	spTSM unadorned literal

Body Lengths

We write $\|b\|$ for the length of b. The metafunction $\|\hat{e}\|$ computes the sum of the lengths of expression literal bodies in $\hat{e}$. It is defined by extending the definition given in Sec.

B.2.1 with the following additional cases:

$$\begin{array}{ll} \|\texttt{implicit syntax}\ \hat{a}\ \texttt{for expressions in}\ \hat{e}\| & = \|\hat{e}\| \\ \|\texttt{implicit syntax}\ \hat{a}\ \texttt{for patterns in}\ \hat{e}\| & = \|\hat{e}\| \\ \|\texttt{/}b\texttt{/}\| & = \|b\| \end{array}$$

Similarly, the metafunction $\|\hat{p}\|$ computes the sum of the lengths of the pattern literal bodies in $\hat{p}$. It is defined by extending the definition given in Sec. B.2.1 with the following additional case:

$$\|\texttt{/}b\texttt{/}\| = \|b\|$$

Textual Syntax

In addition to the stylized syntax, there is also a context-free textual syntax for the UL. We need only posit the existence of the following partial metafunctions.

Condition D.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* $\mathsf{parseUPat}(b) = \hat{p}$*.*

We also impose the following technical conditions.

Condition D.2 (Expression Parsing Monotonicity)**.** *If* $\mathsf{parseUExp}(b) = \hat{e}$ *then* $\|\hat{e}\| < \|b\|$*.*

Condition D.3 (Pattern Parsing Monotonicity)**.** *If* $\mathsf{parseUPat}(b) = \hat{p}$ *then* $\|\hat{p}\| < \|b\|$*.*

D.2.2 Bidirectionally Typed Expansion

Contexts

Unexpanded type formation contexts, $\hat{\Delta}$, and *unexpanded typing contexts*, $\hat{\Gamma}$, were defined in Sec. B.2.3.

Body Encoding and Decoding

The type $\texttt{Body}$ and the judgements $b \downarrow_{\mathsf{Body}} e$ and $e \uparrow_{\mathsf{Body}} b$ are characterized in Sec. B.2.3.

Parse Results

The types $\texttt{ParseResultSE}$ and $\texttt{ParseResultSP}$ are defined as in Sec. B.2.3.

TSM Contexts

seTSM contexts, $\hat{\Psi}$, are of the form $\langle \mathcal{A}; \Psi; \mathcal{I} \rangle$, where $\mathcal{A}$ is a *TSM identifier expansion context*, Ψ is a *seTSM definition context* and $\mathcal{I}$ is a *TSM implicit designation context*.

spTSM contexts, $\hat{\Phi}$, are of the form $\langle \mathcal{A}; \Phi; \mathcal{I} \rangle$, where $\mathcal{A}$ is a TSM identifier expansion context, defined above, and Φ is a *spTSM definition context*.

A *TSM identifier expansion context*, $\mathcal{A}$, is a finite function mapping each TSM identifier $\hat{a} \in \text{dom}(\mathcal{A})$ to the *TSM identifier expansion*, $\hat{a} \rightsquigarrow a$, for some *TSM name*, a. We write $\mathcal{A} \uplus \hat{a} \rightsquigarrow a$ for the TSM identifier expansion context that maps $\hat{a}$ to $\hat{a} \rightsquigarrow a$, and defers to $\mathcal{A}$ for all other TSM identifiers (i.e. the previous mapping is *updated*.)

An *seTSM definition context*, Ψ, is a finite function mapping each TSM name $a \in \text{dom}(\Psi)$ to an *expanded seTSM definition*, $a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}})$, where τ is the seTSM's type annotation, and e_{parse} is its parse function. We write $\Psi, a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}})$ when $a \notin \text{dom}(\Psi)$ for the extension of Ψ that maps a to $a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}})$. We write $\Delta \vdash \Psi\ \mathsf{seTSMs}$ when all the type annotations in Ψ are well-formed assuming Δ, and the parse functions in Ψ are closed and of the appropriate type.

Definition D.4 (seTSM Definition Context Formation). $\Delta \vdash \Psi\ \mathsf{seTSMs}$ *iff for each* $a \hookrightarrow \texttt{setsm}(\tau; e_{parse}) \in \Psi$, *we have* $\Delta \vdash \tau\ \mathsf{type}$ *and* $\emptyset\,\emptyset \vdash e_{parse} : \texttt{parr}(\texttt{Body}; \texttt{ParseResultSE})$.

An *spTSM definition context*, Φ, is a finite function mapping each TSM name $a \in \text{dom}(\Phi)$ to an *expanded seTSM definition*, $a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}})$, where τ is the spTSM's type annotation, and e_{parse} is its parse function. We write $\Phi, a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}})$ when $a \notin \text{dom}(\Phi)$ for the extension of Φ that maps a to $a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}})$. We write $\Delta \vdash \Phi\ \mathsf{spTSMs}$ when all the type annotations in Φ are well-formed assuming Δ, and the parse functions in Φ are closed and of the appropriate type.

Definition D.5 (spTSM Definition Context Formation). $\Delta \vdash \Phi\ \mathsf{spTSMs}$ *iff for each* $a \hookrightarrow \texttt{sptsm}(\tau; e_{parse}) \in \Phi$, *we have* $\Delta \vdash \tau\ \mathsf{type}$ *and* $\emptyset\,\emptyset \vdash e_{parse} : \texttt{parr}(\texttt{Body}; \texttt{ParseResultSP})$.

A *TSM implicit designation context*, $\mathcal{I}$, is a finite function that maps each type $\tau \in \text{dom}(\mathcal{I})$ to the *TSM designation* $\tau \hookrightarrow a$, for some TSM name a. We write $\mathcal{I} \uplus \tau \hookrightarrow a$ for the TSM implicit designation context that maps τ to $\tau \hookrightarrow a$ and defers to $\mathcal{I}$ for all other types (i.e. the previous designation, if any, is updated.)

Definition D.6 (TSM Implicit Designation Context Formation). $\Delta \vdash \mathcal{I}\ \mathsf{designations}$ *iff for each* $\tau \hookrightarrow a \in \mathcal{I}$, *we have* $\Delta \vdash \tau\ \mathsf{type}$.

Definition D.7 (seTSM Context Formation). $\Delta \vdash \langle \mathcal{A}; \Psi; \mathcal{I} \rangle\ \mathsf{seTSMctx}$ *iff*

1. $\Delta \vdash \Psi\ \mathsf{seTSMs}$*; and*
2. *for each* $\hat{a} \rightsquigarrow a \in \mathcal{A}$ *we have* $a \in dom(\Psi)$*; and*
3. $\Delta \vdash \mathcal{I}\ \mathsf{designations}$*; and*
4. *for each* $\tau \hookrightarrow a \in \mathcal{I}$, *we have* $a \in dom(\Psi)$.

Definition D.8 (spTSM Context Formation). $\Delta \vdash \langle \mathcal{A}; \Phi; \mathcal{I} \rangle\ \mathsf{spTSMctx}$ *iff*

1. $\Delta \vdash \Phi\ \mathsf{spTSMs}$*; and*
2. *for each* $\hat{a} \rightsquigarrow a \in \mathcal{A}$ *we have* $a \in dom(\Phi)$*; and*
3. $\Delta \vdash \mathcal{I}\ \mathsf{designations}$*; and*
4. *for each* $\tau \hookrightarrow a \in \mathcal{I}$ *we have* $a \in dom(\Phi)$.

We define $\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}})$, when $\hat{\Psi} = \langle \mathcal{A}; \Phi; \mathcal{I} \rangle$, as an abbreviation of

$$\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}}); \mathcal{I} \rangle$$

We define $\hat{\Phi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}})$, when $\hat{\Phi} = \langle \mathcal{A}; \Phi; \mathcal{I} \rangle$, as an abbreviation of

$$\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}}); \mathcal{I} \rangle$$

Type Expansion

The *type expansion judgement*, $\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau$ type, is inductively defined as in miniVerses$_\mathsf{S}$ by Rules (B.5).

Bidirectionally Typed Expression and Rule Expansion

$\boxed{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau}$ $\hat{e}$ has expansion e synthesizing type τ

$$\frac{}{\hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{x} \rightsquigarrow x \Rightarrow \tau} \tag{D.1a}$$

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \text{ type} \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} : \hat{\tau} \rightsquigarrow e \Rightarrow \tau} \tag{D.1b}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \rightsquigarrow e' \Rightarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{let\,val}\ \hat{x} = \hat{e}\ \mathtt{in}\ \hat{e}' \rightsquigarrow \mathtt{ap(lam\{}\tau\mathtt{\}(}x.e'\mathtt{);}e\mathtt{)} \Rightarrow \tau'} \tag{D.1c}$$

$$\frac{\hat{\Delta} \vdash \hat{\tau}_1 \rightsquigarrow \tau_1 \text{ type} \qquad \hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau_1 \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau_2}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \lambda \hat{x}{:}\hat{\tau}_1.\hat{e} \rightsquigarrow \mathtt{lam\{}\tau_1\mathtt{\}(}x.e\mathtt{)} \Rightarrow \mathtt{parr(}\tau_1\mathtt{;}\tau_2\mathtt{)}} \tag{D.1d}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_1 \rightsquigarrow e_1 \Rightarrow \mathtt{parr(}\tau_2\mathtt{;}\tau\mathtt{)} \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_2 \rightsquigarrow e_2 \Leftarrow \tau_2}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_1(\hat{e}_2) \rightsquigarrow \mathtt{ap(}e_1\mathtt{;}e_2\mathtt{)} \Rightarrow \tau} \tag{D.1e}$$

$$\frac{\hat{\Delta}, \hat{t} \rightsquigarrow t \text{ type}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \Lambda \hat{t}.\hat{e} \rightsquigarrow \mathtt{tlam(}t.e\mathtt{)} \Rightarrow \mathtt{all(}t.\tau\mathtt{)}} \tag{D.1f}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \mathtt{all(}t.\tau\mathtt{)} \qquad \hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau' \text{ type}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}\mathtt{[}\hat{\tau}'\mathtt{]} \rightsquigarrow \mathtt{tap\{}\tau'\mathtt{\}(}e\mathtt{)} \Rightarrow [\tau'/t]\tau} \tag{D.1g}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \mathtt{rec(}t.\tau\mathtt{)}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{unfold(}\hat{e}\mathtt{)} \rightsquigarrow \mathtt{unfold(}e\mathtt{)} \Rightarrow [\mathtt{rec(}t.\tau\mathtt{)}/t]\tau} \tag{D.1h}$$

$$\frac{\{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_i \rightsquigarrow e_i \Rightarrow \tau_i\}_{i \in L}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \langle\{i \hookrightarrow \hat{e}_i\}_{i \in L}\rangle \rightsquigarrow \mathtt{tpl}[L]\mathtt{(}\{i \hookrightarrow e_i\}_{i \in L}\mathtt{)} \Rightarrow \mathtt{prod}[L]\mathtt{(}\{i \hookrightarrow \tau_i\}_{i \in L}\mathtt{)}} \tag{D.1i}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \mathtt{prod}[L, \ell]\mathtt{(}\{i \hookrightarrow \tau_i\}_{i \in L}\mathtt{;}\ell \hookrightarrow \tau\mathtt{)}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \cdot \ell \rightsquigarrow \mathtt{prj}[\ell]\mathtt{(}e\mathtt{)} \Rightarrow \tau} \tag{D.1j}$$

$$\frac{\begin{array}{cc} \hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \text{ type} & \emptyset\ \emptyset \vdash e_{\text{parse}} : \mathtt{parr(Body;ParseResultSE)} \\ e_{\text{parse}} \Downarrow e'_{\text{parse}} & \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{setsm}(\tau; e'_{\text{parse}});\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau' \end{array}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{for\ expressions}\ \{e_{\text{parse}}\}\ \mathtt{in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \tag{D.1k}$$

$$\frac{\begin{array}{c}\hat{\Psi} = \hat{\Psi}', \hat{a} \rightsquigarrow a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}}) \\ b \downarrow_{\mathsf{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}[\texttt{SuccessE}](e_{\text{proto}}) \qquad e_{\text{proto}} \uparrow_{\mathsf{PrExpr}} \grave{e} \\ \mathsf{seg}(\grave{e}) \text{ segments } b \qquad \emptyset\,\emptyset \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e \Leftarrow \tau\end{array}}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{a}\ \text{`}b\text{`} \rightsquigarrow e \Rightarrow \tau} \tag{D.1l}$$

$$\frac{\begin{array}{c}\hat{\Psi} = \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}}); \mathcal{I}\rangle \\ \hat{\Delta}\,\hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a\rangle;\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau'\end{array}}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{implicit syntax}\ \hat{a}\ \texttt{for expressions in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \tag{D.1m}$$

$$\frac{\begin{array}{c}\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \text{ type} \qquad \emptyset\,\emptyset \vdash e_{\text{parse}} : \texttt{parr}(\texttt{Body};\texttt{ParseResultSP}) \\ e_{\text{parse}} \Downarrow e'_{\text{parse}} \qquad \hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{sptsm}(\tau; e'_{\text{parse}})} \hat{e} \rightsquigarrow e \Rightarrow \tau'\end{array}}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for patterns by static}\ e_{\text{parse}}\ \texttt{in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \tag{D.1n}$$

$$\frac{\begin{array}{c}\hat{\Phi} = \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}}); \mathcal{I}\rangle \\ \hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a\rangle} \hat{e} \rightsquigarrow e \Rightarrow \tau'\end{array}}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{implicit syntax}\ \hat{a}\ \texttt{for patterns in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \tag{D.1o}$$

$\boxed{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau}$ $\hat{e}$ has expansion e when analyzed against type τ

$$\frac{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau} \tag{D.2a}$$

$$\frac{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \hat{\Delta}\,\hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \rightsquigarrow e' \Leftarrow \tau'}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{let val}\ \hat{x} = \hat{e}\ \texttt{in}\ \hat{e}' \rightsquigarrow \texttt{ap}(\texttt{lam}\{\tau\}(x.e');e) \Leftarrow \tau'} \tag{D.2b}$$

$$\frac{\hat{\Delta}, \hat{t} \rightsquigarrow t \text{ type}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \Lambda\hat{t}.\hat{e} \rightsquigarrow \texttt{tlam}(t.e) \Leftarrow \texttt{all}(t.\tau)} \tag{D.2c}$$

$$\frac{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow [\texttt{rec}(t.\tau)/t]\tau}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{fold}(\hat{e}) \rightsquigarrow \texttt{fold}(e) \Leftarrow \texttt{rec}(t.\tau)} \tag{D.2d}$$

$$\frac{\{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_i \rightsquigarrow e_i \Leftarrow \tau_i\}_{i \in L}}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \langle\{i \hookrightarrow \hat{e}_i\}_{i \in L}\rangle \rightsquigarrow \texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) \Leftarrow \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{D.2e}$$

$$\frac{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{inj}[\ell](\hat{e}) \rightsquigarrow \texttt{inj}[\ell](e) \Leftarrow \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau')} \tag{D.2f}$$

$$\frac{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \rightsquigarrow r_i \Leftarrow \tau \Mapsto \tau'\}_{1 \le i \le n}}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{match}\ \hat{e}\ \{\hat{r}_i\}_{1 \le i \le n} \rightsquigarrow \texttt{match}[n](e;\{r_i\}_{1 \le i \le n}) \Leftarrow \tau'} \tag{D.2g}$$

$$\frac{\hat{\Delta} \vdash \hat{\tau} \leadsto \tau \ \mathsf{type} \qquad \emptyset\, \emptyset \vdash e_{\text{parse}} : \mathtt{parr(Body;ParseResultSE)} \qquad e_{\text{parse}} \Downarrow e'_{\text{parse}} \qquad \hat{\Delta}\, \hat{\Gamma} \vdash_{\hat{\Psi}, \hat{a} \leadsto a \hookrightarrow \mathtt{setsm}(\tau; e'_{\text{parse}}); \hat{\Phi}} \hat{e} \leadsto e \Leftarrow \tau'}{\hat{\Delta}\, \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{for\ expressions}\ \{e_{\text{parse}}\}\ \mathtt{in}\ \hat{e} \leadsto e \Leftarrow \tau'} \tag{D.2h}$$

$$\frac{\hat{\Psi} = \langle \mathcal{A} \uplus \hat{a} \leadsto a; \Psi, a \hookrightarrow \mathtt{setsm}(\tau; e_{\text{parse}}); \mathcal{I} \rangle \qquad \hat{\Delta}\, \hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \leadsto a; \Psi, a \hookrightarrow \mathtt{setsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle; \hat{\Phi}} \hat{e} \leadsto e \Leftarrow \tau'}{\hat{\Delta}\, \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \mathtt{implicit\ syntax}\ \hat{a}\ \mathtt{for\ expressions\ in}\ \hat{e} \leadsto e \Leftarrow \tau'} \tag{D.2i}$$

$$\frac{\begin{array}{c}\hat{\Psi} = \langle \mathcal{A}; \Psi, a \hookrightarrow \mathtt{setsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle \\ b \downarrow_{\mathsf{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \mathtt{inj[SuccessE]}(e_{\text{proto}}) \qquad e_{\text{proto}} \uparrow_{\mathsf{PrExpr}} \grave{e} \\ \mathsf{seg}(\grave{e})\ \mathsf{segments}\ b \qquad \emptyset\, \emptyset \vdash^{\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; \hat{\Phi}; b} \grave{e} \leadsto e \Leftarrow \tau\end{array}}{\hat{\Delta}\, \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} /b/ \leadsto e \Leftarrow \tau} \tag{D.2j}$$

$$\frac{\hat{\Delta} \vdash \hat{\tau} \leadsto \tau \ \mathsf{type} \qquad \emptyset\, \emptyset \vdash e_{\text{parse}} : \mathtt{parr(Body;ParseResultSP)} \qquad e_{\text{parse}} \Downarrow e'_{\text{parse}} \qquad \hat{\Delta}\, \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}, \hat{a} \leadsto a \hookrightarrow \mathtt{sptsm}(\tau; e'_{\text{parse}})} \hat{e} \leadsto e \Leftarrow \tau'}{\hat{\Delta}\, \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{for\ patterns\ by\ static}\ e_{\text{parse}}\ \mathtt{in}\ \hat{e} \leadsto e \Leftarrow \tau'} \tag{D.2k}$$

$$\frac{\hat{\Phi} = \langle \mathcal{A} \uplus \hat{a} \leadsto a; \Phi, a \hookrightarrow \mathtt{sptsm}(\tau; e_{\text{parse}}); \mathcal{I} \rangle \qquad \hat{\Delta}\, \hat{\Gamma} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \leadsto a; \Phi, a \hookrightarrow \mathtt{sptsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle} \hat{e} \leadsto e \Leftarrow \tau'}{\hat{\Delta}\, \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \mathtt{implicit\ syntax}\ \hat{a}\ \mathtt{for\ patterns\ in}\ \hat{e} \leadsto e \Leftarrow \tau'} \tag{D.2l}$$

$\boxed{\hat{\Delta}\, \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{r} \leadsto r \Leftarrow \tau \Mapsto \tau'}$ $\hat{r}$ has expansion r taking values of type τ to values of type τ'

$$\frac{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \leadsto p : \tau \dashv\!\vert \langle \mathcal{G}'; \Gamma' \rangle \qquad \langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G} \uplus \mathcal{G}'; \Gamma \cup \Gamma' \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \leadsto e \Leftarrow \tau'}{\langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{p} \Rightarrow \hat{e} \leadsto \mathtt{rule}(p.e) \Leftarrow \tau \Mapsto \tau'} \tag{D.3}$$

Pattern Expansion

$\boxed{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \leadsto p : \tau \dashv\!\vert \hat{\Gamma}}$ $\hat{p}$ has expansion p matching against τ generating hypotheses $\hat{\Gamma}$

$$\frac{}{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{x} \leadsto x : \tau \dashv\!\vert \langle \hat{x} \leadsto x; x : \tau \rangle} \tag{D.4a}$$

$$\frac{}{\hat{\Delta} \vdash_{\hat{\Phi}} \_ \leadsto \mathtt{wildp} : \tau \dashv\!\vert \langle \emptyset; \emptyset \rangle} \tag{D.4b}$$

$$\frac{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \leadsto p : [\mathtt{rec}(t.\tau)/t]\tau \dashv\!\vert \hat{\Gamma}}{\hat{\Delta} \vdash_{\hat{\Phi}} \mathtt{fold}(\hat{p}) \leadsto \mathtt{foldp}(p) : \mathtt{rec}(t.\tau) \dashv\!\vert \hat{\Gamma}} \tag{D.4c}$$

$$\frac{\tau = \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \qquad \{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p}_i \leadsto p_i : \tau_i \dashv\!\vert \hat{\Gamma}_i\}_{i \in L}}{\hat{\Delta} \vdash_{\hat{\Phi}} \langle \{i \hookrightarrow \hat{p}_i\}_{i \in L} \rangle \leadsto \mathtt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \tau \dashv\!\vert \uplus_{i \in L} \hat{\Gamma}_i} \tag{D.4d}$$

$$\frac{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \hat{\Gamma}}{\hat{\Delta} \vdash_{\hat{\Phi}} \texttt{inj}[\ell](\hat{p}) \rightsquigarrow \texttt{injp}[\ell](p) : \texttt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i\in L}; \ell \hookrightarrow \tau) \dashv\!\mid \hat{\Gamma}} \quad \text{(D.4e)}$$

$$\frac{\begin{array}{c}\hat{\Phi} = \hat{\Phi}', \hat{a} \rightsquigarrow a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}}) \\ b \downarrow_{\mathsf{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}[\texttt{SuccessP}](e_{\text{proto}}) \qquad e_{\text{proto}} \uparrow_{\mathsf{PrPat}} \grave{p} \\ \mathsf{seg}(\grave{p}) \mathsf{~segments~} b \qquad \grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\hat{\Delta};\hat{\Phi};b} \hat{\Gamma}\end{array}}{\hat{\Delta} \vdash_{\hat{\Phi}} \hat{a}\ \text{`}b\text{`} \rightsquigarrow p : \tau \dashv\!\mid \hat{\Gamma}} \quad \text{(D.4f)}$$

$$\frac{\begin{array}{c}\hat{\Phi} = \langle \mathcal{A}; \Phi, a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}}); \mathcal{I}, \tau \hookrightarrow a\rangle \\ b \downarrow_{\mathsf{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}[\texttt{SuccessP}](e_{\text{proto}}) \qquad e_{\text{proto}} \uparrow_{\mathsf{PrPat}} \grave{p} \\ \mathsf{seg}(\grave{p}) \mathsf{~segments~} b \qquad \grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\hat{\Delta};\hat{\Phi};b} \hat{\Gamma}\end{array}}{\hat{\Delta} \vdash_{\hat{\Phi}} /b/ \rightsquigarrow p : \tau \dashv\!\mid \hat{\Gamma}} \quad \text{(D.4g)}$$

D.3 Proto-Expansion Validation

D.3.1 Syntax of Proto-Expansions

The syntax of proto-expansions was defined in Sec. B.3.

Common Proto-Expansion Terms

Each expanded term, except variable patterns, maps onto a proto-expansion term. We refer to these as the *common proto-expansion terms*. In particular:

- Each type, τ, maps onto a proto-type, $\mathcal{P}(\tau)$, as follows:

$$\begin{aligned}
\mathcal{P}(t) &= t \\
\mathcal{P}(\texttt{parr}(\tau_1;\tau_2)) &= \texttt{prparr}(\mathcal{P}(\tau_1);\mathcal{P}(\tau_2)) \\
\mathcal{P}(\texttt{all}(t.\tau)) &= \texttt{prall}(t.\mathcal{P}(\tau)) \\
\mathcal{P}(\texttt{rec}(t.\tau)) &= \texttt{prrec}(t.\mathcal{P}(\tau)) \\
\mathcal{P}(\texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})) &= \texttt{prprod}[L](\{i \hookrightarrow \mathcal{P}(\tau_i)\}_{i\in L}) \\
\mathcal{P}(\texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i\in L})) &= \texttt{prsum}[L](\{i \hookrightarrow \mathcal{P}(\tau_i)\}_{i\in L})
\end{aligned}$$

- Each expanded expression, e, maps onto a proto-expression, $\mathcal{P}(e)$, as follows:

$$\begin{aligned}
\mathcal{P}(x) &= x \\
\mathcal{P}(\texttt{lam}\{\tau\}(x.e)) &= \texttt{prlam}\{\mathcal{P}(\tau)\}(x.\mathcal{P}(e)) \\
\mathcal{P}(\texttt{ap}(e_1;e_2)) &= \texttt{prap}(\mathcal{P}(e_1);\mathcal{P}(e_2)) \\
\mathcal{P}(\texttt{tlam}(t.e)) &= \texttt{prtlam}(t.\mathcal{P}(e)) \\
\mathcal{P}(\texttt{tap}\{\tau\}(e)) &= \texttt{prtap}\{\mathcal{P}(\tau)\}(\mathcal{P}(e)) \\
\mathcal{P}(\texttt{fold}(e)) &= \texttt{prasc}\{\texttt{prrec}(t.\mathcal{P}(\tau))\}(\texttt{prfold}(\mathcal{P}(e))) \\
\mathcal{P}(\texttt{unfold}(e)) &= \texttt{prunfold}(\mathcal{P}(e)) \\
\mathcal{P}(\texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L})) &= \texttt{prtpl}\{L\}(\{i \hookrightarrow \mathcal{P}(e_i)\}_{i\in L}) \\
\mathcal{P}(\texttt{inj}[\ell](e)) &= \texttt{prasc}\{\texttt{prsum}[L](\{i \hookrightarrow \mathcal{P}(\tau_i)\}_{i\in L})\}(\texttt{prinj}[\ell](\mathcal{P}(e))) \\
\mathcal{P}(\texttt{match}[n](e;\{r_i\}_{1\le i\le n})) &= \texttt{prasc}\{\mathcal{P}(\tau)\}(\texttt{prmatch}[n](\mathcal{P}(e);\{\mathcal{P}(r_i)\}_{1\le i\le n}))
\end{aligned}$$

- Each expanded rule, r, maps onto the proto-rule, $\mathcal{P}(r)$, as follows:

$$\mathcal{P}(\texttt{rule}(p.e)) = \texttt{prrule}(p.\mathcal{P}(e))$$

- Each expanded pattern, p, except for the variable patterns, maps onto a proto-pattern, $\mathcal{P}(p)$, as follows:

$$\begin{aligned}
\mathcal{P}(\texttt{wildp}) &= \texttt{prwildp} \\
\mathcal{P}(\texttt{foldp}(p)) &= \texttt{prfoldp}(\mathcal{P}(p)) \\
\mathcal{P}(\texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L})) &= \texttt{prtplp}[L](\{i \hookrightarrow \mathcal{P}(p_i)\}_{i \in L}) \\
\mathcal{P}(\texttt{injp}[\ell](p)) &= \texttt{prinjp}[\ell](\mathcal{P}(p))
\end{aligned}$$

These definitions differ from those given in Sec. B.3 in that they include the type information necessary for bidirectional typechecking.

Proto-Expression Encoding and Decoding

The type $\texttt{PrExpr}$ and the judgements $\grave{e} \downarrow_{\mathsf{PrExpr}} e$ and $e \uparrow_{\mathsf{PrExpr}} \grave{e}$ are characterized as described in Sec. B.3.

Proto-Pattern Encoding and Decoding

The type $\texttt{PrPat}$ and the judgements $\grave{p} \downarrow_{\mathsf{PrPat}} e$ and $e \uparrow_{\mathsf{PrPat}} \grave{p}$ are characterized as described in Sec. B.3.

Splice Summaries

The *splice summary* of a proto-expression, $\mathsf{summary}(\grave{e})$, or proto-pattern, $\mathsf{summary}(\grave{p})$, is the finite set of references to spliced types, expressions and patterns that it mentions.

Segmentations

A *segment set*, ψ, is a finite set of pairs of natural numbers indicating the locations of spliced terms. The *segmentation* of a proto-expression, $\mathsf{seg}(\grave{e})$, or proto-pattern, $\mathsf{seg}(\grave{p})$, is the segment set implied by its splice summary.

D.3.2 Proto-Expansion Validation

Proto-Type Validation

The *proto-type validation judgement*, $\Delta \vdash^{\mathbb{T}} \grave{\tau} \rightsquigarrow \tau\ \mathsf{type}$, is inductively defined by Rules (B.9), which were defined in Sec. B.3.2.

Bidirectional Proto-Expression and Proto-Rule Validation

Expression splicing scenes, $\mathbb{E}$, are of the form $\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; \hat{\Phi}; b$. We write $\mathsf{ts}(\mathbb{E})$ for the type splicing scene constructed by dropping unnecessary contexts from $\mathbb{E}$:

$$\mathsf{ts}(\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; \hat{\Phi}; b) = \hat{\Delta}; b$$

$\boxed{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau}$ $\grave{e}$ has expansion e synthesizing type τ

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash^{\mathbb{E}} x \rightsquigarrow x \Rightarrow \tau} \tag{D.5a}$$

$$\frac{\Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prasc}\{\grave{\tau}\}(\grave{e}) \rightsquigarrow e \Rightarrow \tau} \tag{D.5b}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \Delta\ \Gamma, x : \tau \vdash^{\mathbb{E}} \grave{e}' \rightsquigarrow e' \Rightarrow \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prletval}(\grave{e}; x.\grave{e}') \rightsquigarrow \mathtt{ap}(\mathtt{lam}\{\tau\}(x.e'); e) \Rightarrow \tau'} \tag{D.5c}$$

$$\frac{\Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau}_1 \rightsquigarrow \tau_1\ \mathsf{type} \qquad \Delta\ \Gamma, x : \tau_1 \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau_2}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prlam}\{\grave{\tau}_1\}(x.\grave{e}) \rightsquigarrow \mathtt{lam}\{\tau_1\}(x.e) \Rightarrow \mathtt{parr}(\tau_1; \tau_2)} \tag{D.5d}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e}_1 \rightsquigarrow e_1 \Rightarrow \mathtt{parr}(\tau_2; \tau) \qquad \Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e}_2 \rightsquigarrow e_2 \Leftarrow \tau_2}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prap}(\grave{e}_1; \grave{e}_2) \rightsquigarrow \mathtt{ap}(e_1; e_2) \Rightarrow \tau} \tag{D.5e}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prtlam}(t.\grave{e}) \rightsquigarrow \mathtt{tlam}(t.e) \Rightarrow \mathtt{all}(t.\tau)} \tag{D.5f}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \mathtt{all}(t.\tau) \qquad \Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau}' \rightsquigarrow \tau'\ \mathsf{type}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prtap}\{\grave{\tau}'\}(\grave{e}) \rightsquigarrow \mathtt{tap}\{\tau'\}(e) \Rightarrow [\tau'/t]\tau} \tag{D.5g}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \mathtt{rec}(t.\tau)}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prunfold}(\grave{e}) \rightsquigarrow \mathtt{unfold}(e) \Rightarrow [\mathtt{rec}(t.\tau)/t]\tau} \tag{D.5h}$$

$$\frac{\begin{array}{c}\tau = \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \\ \{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e}_i \rightsquigarrow e_i \Rightarrow \tau_i\}_{i \in L}\end{array}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prtpl}\{L\}(\{i \hookrightarrow \grave{e}_i\}_{i \in L}) \rightsquigarrow \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) \Rightarrow \tau} \tag{D.5i}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \mathtt{prod}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prprj}[\ell](\grave{e}) \rightsquigarrow \mathtt{prj}[\ell](e) \Rightarrow \tau} \tag{D.5j}$$

$$\frac{\begin{array}{c}\emptyset \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \mathbb{E} = \langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle; \langle \mathcal{G}; \Gamma_{\mathsf{app}} \rangle; \hat{\Psi}; \hat{\Phi}; b \\ \mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e} \qquad \langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle\ \langle \mathcal{G}; \Gamma_{\mathsf{app}} \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau \\ \Delta \cap \Delta_{\mathsf{app}} = \emptyset \qquad \mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma_{\mathsf{app}}) = \emptyset\end{array}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{splicede}[m; n; \grave{\tau}] \rightsquigarrow e \Rightarrow \tau} \tag{D.5k}$$

$\boxed{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau}$ $\grave{e}$ has expansion e when analyzed against type τ

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau} \tag{D.6a}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \Delta\ \Gamma, x : \tau \vdash^{\mathbb{E}} \grave{e}' \rightsquigarrow e' \Leftarrow \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prletval}(\grave{e}; x.\grave{e}') \rightsquigarrow \mathtt{ap}(\mathtt{lam}\{\tau\}(x.e'); e) \Leftarrow \tau'} \tag{D.6b}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prtlam}(t.\grave{e}) \rightsquigarrow \mathtt{tlam}(t.e) \Leftarrow \mathtt{all}(t.\tau)} \tag{D.6c}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow [\mathtt{rec}(t.\tau)/t]\tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prfold}(\grave{e}) \rightsquigarrow \mathtt{fold}(e) \Leftarrow \mathtt{rec}(t.\tau)} \tag{D.6d}$$

$$\frac{\begin{array}{c}\tau = \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \\ \{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e}_i \rightsquigarrow e_i \Leftarrow \tau_i\}_{i \in L}\end{array}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prtpl}\{L\}(\{i \hookrightarrow \grave{e}_i\}_{i \in L}) \rightsquigarrow \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) \Leftarrow \tau} \tag{D.6e}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prinj}[\ell](\grave{e}) \rightsquigarrow \mathtt{inj}[\ell](e) \Leftarrow \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau')} \tag{D.6f}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r}_i \rightsquigarrow r_i \Leftarrow \tau \Mapsto \tau'\}_{1 \leq i \leq n}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prmatch}[n](\grave{e}; \{\grave{r}_i\}_{1 \leq i \leq n}) \rightsquigarrow \mathtt{match}[n](e; \{r_i\}_{1 \leq i \leq n}) \Leftarrow \tau'} \tag{D.6g}$$

$\boxed{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r} \rightsquigarrow r \Leftarrow \tau \Mapsto \tau'}$ $\grave{r}$ has expansion r taking values of type τ to values of type τ'

$$\frac{\Delta \vdash p : \tau \dashv\!\mid \Gamma \qquad \Delta\ \Gamma \cup \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prrule}(p.\grave{e}) \rightsquigarrow \mathtt{rule}(p.e) \Leftarrow \tau \Mapsto \tau'} \tag{D.7}$$

Proto-Pattern Validation

Pattern splicing scenes, $\mathbb{P}$, are of the form $\hat{\Delta}; \hat{\Phi}; b$.

$\boxed{\grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\mathbb{P}} \hat{\Gamma}}$ $\grave{p}$ has expansion p matching against τ generating hypotheses $\hat{\Gamma}$

$$\frac{}{\mathtt{prwildp} \rightsquigarrow \mathtt{wildp} : \tau \dashv\!\mid^{\mathbb{P}} \langle\emptyset; \emptyset\rangle} \tag{D.8a}$$

$$\frac{\grave{p} \rightsquigarrow p : [\mathtt{rec}(t.\tau)/t]\tau \dashv\!\mid^{\mathbb{P}} \hat{\Gamma}}{\mathtt{prfoldp}(\grave{p}) \rightsquigarrow \mathtt{foldp}(p) : \mathtt{rec}(t.\tau) \dashv\!\mid^{\mathbb{P}} \hat{\Gamma}} \tag{D.8b}$$

$$\frac{\begin{array}{c}\tau = \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \\ \{\grave{p}_i \rightsquigarrow p_i : \tau_i \dashv\!\mid^{\mathbb{P}} \hat{\Gamma}_i\}_{i \in L}\end{array}}{\mathtt{prtplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i \in L}) \rightsquigarrow \mathtt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \tau \dashv\!\mid^{\mathbb{P}} \uplus_{i \in L} \hat{\Gamma}_i} \tag{D.8c}$$

$$\frac{\grave{p} \leadsto p : \tau \dashv\!\!|^{\mathbb{P}} \hat{\Gamma}}{\texttt{prinjp}[\ell](\grave{p}) \leadsto \texttt{injp}[\ell](p) : \texttt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\!\!|^{\mathbb{P}} \hat{\Gamma}} \quad \text{(D.8d)}$$

$$\frac{\emptyset \vdash^{\hat{\Delta}; b} \grave{\tau} \leadsto \tau \ \mathsf{type} \qquad \mathsf{parseUPat}(\mathsf{subseq}(b; m; n)) = \hat{p} \qquad \hat{\Delta} \vdash_{\hat{\Phi}} \hat{p} \leadsto p : \tau \dashv\!\!| \hat{\Gamma}}{\texttt{splicedp}[m; n; \grave{\tau}] \leadsto p : \tau \dashv\!\!|^{\hat{\Delta}; \hat{\Phi}; b} \hat{\Gamma}} \quad \text{(D.8e)}$$

D.4 Metatheory

D.4.1 Typed Pattern Expansion

Theorem D.9 (Typed Pattern Expansion).

1. *If* $\langle \mathcal{D}; \Delta \rangle \vdash_{\langle \mathcal{A}; \Phi; \mathcal{I} \rangle} \hat{p} \leadsto p : \tau \dashv\!\!| \langle \mathcal{G}; \Gamma \rangle$ *then* $\Delta \vdash p : \tau \dashv\!\!| \Gamma$.
2. *If* $\grave{p} \leadsto p : \tau \dashv\!\!|^{\langle \mathcal{D}; \Delta \rangle; \langle \mathcal{A}; \Phi \rangle; b} \langle \mathcal{G}; \Gamma \rangle$ *then* $\Delta \vdash p : \tau \dashv\!\!| \Gamma$.

Proof. By mutual rule induction over Rules (D.4) and Rules (D.8

1. We induct on the premise. In the following, let $\hat{\Delta} = \langle \mathcal{D}; \Delta \rangle$ and $\hat{\Gamma} = \langle \mathcal{G}; \Gamma \rangle$ and $\hat{\Phi} = \langle \mathcal{A}; \Phi; \mathcal{I} \rangle$.

   Case (D.4a) **through** (D.4f)**.** These cases follow by identical argument to the corresponding cases of Theorem B.26.

   Case (D.4g)**.**

(1)	$\hat{p} = /b/$	by assumption
(2)	$\Phi = \Phi', a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}})$	by assumption
(3)	$\mathcal{I} = \mathcal{I}', \tau \hookrightarrow a$	by assumption
(4)	$b \downarrow_{\mathsf{Body}} e_{\text{body}}$	by assumption
(5)	$e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}[\texttt{SuccessP}](e_{\text{proto}})$	by assumption
(6)	$e_{\text{proto}} \uparrow_{\mathsf{PrPat}} \grave{p}$	by assumption
(7)	$\grave{p} \leadsto p : \tau \dashv\!\!|^{\hat{\Delta}; \hat{\Phi}; b} \hat{\Gamma}$	by assumption
(8)	$\Delta \vdash p : \tau \dashv\!\!| \Gamma$	by IH, part 2 on (7)

2. We induct on the premise. All cases follow by identical argument to the corresponding cases of Theorem B.26.

The mutual induction can be shown to be well-founded by an argument nearly identical to that that given in the proof of Theorem B.26, differing only in that the appeal to Condition B.13 is replaced by an appeal to the analagous Condition D.3.

□

D.4.2 Typed Expression and Rule Expansion

Theorem D.10 (Typed Expression and Rule Expansion).

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

(c) *If* $\langle \mathcal{D}; \Delta \rangle \; \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{r} \rightsquigarrow r \Leftarrow \tau \Mapsto \tau'$ *and* $\Delta \vdash \tau'$ type *then* $\Delta \; \Gamma \vdash r : \tau \Mapsto \tau'$.

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

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

   (c) *If* $\Delta \; \Gamma \vdash^{\langle \mathcal{D}; \Delta_{app} \rangle; \langle \mathcal{G}; \Gamma_{app} \rangle; \hat{\Psi}; \hat{\Phi}; b} \grave{r} \rightsquigarrow r \Leftarrow \tau \Mapsto \tau'$ *and* $\Delta \vdash \tau'$ type *and* $\Delta \cap \Delta_{app} = \emptyset$ *and* $dom(\Gamma) \cap dom(\Gamma_{app}) = \emptyset$ *then* $\Delta \cup \Delta_{app} \; \Gamma \cup \Gamma_{app} \vdash r : \tau \Mapsto \tau'$.

Proof. By mutual rule induction over Rules (D.1), Rules (D.2), Rule (D.3), Rules (D.5), Rules (D.6) and Rule (D.7). The proof follows the proof of Theorem B.27. □

D.4.3 Abstract Reasoning Principles

Lemma D.11 (Proto-Expression and Proto-Rule Expansion Decomposition)**.**

1. *If* ($\Delta \; \Gamma \vdash^{\langle \mathcal{D}; \Delta_{app} \rangle; \langle \mathcal{G}; \Gamma_{app} \rangle; \hat{\Psi}; \hat{\Phi}; b} \grave{e} \rightsquigarrow e \Leftarrow \tau$ *or* $\Delta \; \Gamma \vdash^{\langle \mathcal{D}; \Delta_{app} \rangle; \langle \mathcal{G}; \Gamma_{app} \rangle; \hat{\Psi}; \hat{\Phi}; b} \grave{e} \rightsquigarrow e \Rightarrow \tau$) *where* $\mathsf{summary}(\grave{e}) = \{\mathtt{splicedt}[m'_i; n'_i]\}_{0 \le i < n_{ty}} \cup \{\mathtt{splicede}[m_i; n_i; \grave{\tau}_i]\}_{0 \le i < n_{exp}}$ *then all of the following hold:*
   (a) $\{\langle \mathcal{D}; \Delta_{app} \rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow \tau'_i \; \mathsf{type}\}_{0 \le i < n_{ty}}$
   (b) $\{\emptyset \vdash^{\langle \mathcal{D}; \Delta_{app} \rangle; b} \grave{\tau}_i \rightsquigarrow \tau_i \; \mathsf{type}\}_{0 \le i < n_{exp}}$
   (c) $\{\langle \mathcal{D}; \Delta_{app} \rangle \; \langle \mathcal{G}; \Gamma_{app} \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \mathsf{parseUExp}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow e_i \Leftarrow \tau_i\}_{0 \le i < n_{exp}}$
   (d) $e = [\{\tau'_i / t_i\}_{0 \le i < n_{ty}}, \{e_i / x_i\}_{0 \le i < n_{exp}}]e'$ *for some* e' *and* $\{t_i\}_{0 \le i < n_{ty}}$ *and* $\{x_i\}_{0 \le i < n_{exp}}$ *such that* $\{t_i\}_{0 \le i < n_{ty}}$ *fresh (i.e.* $\{t_i \notin dom(\Delta)\}_{0 \le i < n_{ty}}$ *and* $\{t_i \notin dom(\Delta_{app})\}_{0 \le i < n_{ty}}$*) and* $\{x_i\}_{0 \le i < n_{exp}}$ *fresh (i.e.* $\{x_i \notin dom(\Gamma)\}_{0 \le i < n_{exp}}$ *and* $\{x_i \notin dom(\Gamma_{app})\}_{0 \le i < n_{ty}}$*)*
   (e) $\mathsf{fv}(e') \subset dom(\Delta) \cup dom(\Gamma) \cup \{t_i\}_{0 \le i < n_{ty}} \cup \{x_i\}_{0 \le i < n_{exp}}$
2. *If* $\Delta \; \Gamma \vdash^{\langle \mathcal{D}; \Delta_{app} \rangle; \langle \mathcal{G}; \Gamma_{app} \rangle; \hat{\Psi}; \hat{\Phi}; b} \grave{r} \rightsquigarrow r : \tau \Mapsto \tau'$ *where*

$$\mathsf{summary}(\grave{r}) = \{\mathtt{splicedt}[m'_i; n'_i]\}_{0 \le i < n_{ty}} \cup \{\mathtt{splicede}[m_i; n_i; \grave{\tau}_i]\}_{0 \le i < n_{exp}}$$

   then all of the following hold:
   (a) $\{\langle \mathcal{D}; \Delta_{app} \rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow \tau'_i \; \mathsf{type}\}_{0 \le i < n_{ty}}$
   (b) $\{\emptyset \vdash^{\langle \mathcal{D}; \Delta_{app} \rangle; b} \grave{\tau}_i \rightsquigarrow \tau_i \; \mathsf{type}\}_{0 \le i < n_{exp}}$
   (c) $\{\langle \mathcal{D}; \Delta_{app} \rangle \; \langle \mathcal{G}; \Gamma_{app} \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \mathsf{parseUExp}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow e_i \Leftarrow \tau_i\}_{0 \le i < n_{exp}}$
   (d) $r = [\{\tau'_i / t_i\}_{0 \le i < n_{ty}}, \{e_i / x_i\}_{0 \le i < n_{exp}}]r'$ *for some* e' *and fresh* $\{t_i\}_{0 \le i < n_{ty}}$ *and fresh* $\{x_i\}_{0 \le i < n_{exp}}$
   (e) $\mathsf{fv}(r') \subset dom(\Delta) \cup dom(\Gamma) \cup \{t_i\}_{0 \le i < n_{ty}} \cup \{x_i\}_{0 \le i < n_{exp}}$

Proof. By rule induction over Rules (D.6) and Rule (D.7). The proof follows the proof of Theorem B.30. □

Theorem D.12 (seTSM Abstract Reasoning Principles - Explicit Application)**.** *If*

$$\langle \mathcal{D}; \Delta \rangle \; \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{a} \; \text{`} b \text{`} \rightsquigarrow e \Rightarrow \tau$$

then:

1. *(**Typing 1**)* $\hat{\Psi} = \hat{\Psi}', \hat{a} \rightsquigarrow a \hookrightarrow \texttt{setsm}(\tau; e_{parse})$ *and* $\Delta\ \Gamma \vdash e : \tau$
2. $b \downarrow_{\mathsf{Body}} e_{body}$
3. $e_{parse}(e_{body}) \Downarrow \texttt{inj}[\mathtt{SuccessE}](e_{proto})$
4. $e_{proto} \uparrow_{\mathsf{PrExpr}} \grave{e}$
5. *(**Segmentation**)* $\mathsf{seg}(\grave{e})$ segments b
6. $\mathsf{summary}(\grave{e}) = \{\texttt{splicedt}[m'_i; n'_i]\}_{0 \leq i < n_{ty}} \cup \{\texttt{splicede}[m_i; n_i; \grave{\tau}_i]\}_{0 \leq i < n_{exp}}$
7. *(**Typing 2**)* $\{\langle \mathcal{D}; \Delta \rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow \tau'_i\ \mathsf{type}\}_{0 \leq i < n_{ty}}$ *and* $\{\Delta \vdash \tau'_i\ \mathsf{type}\}_{0 \leq i < n_{ty}}$
8. *(**Typing 3**)* $\{\emptyset \vdash^{\langle \mathcal{D}; \Delta \rangle; b} \grave{\tau}_i \rightsquigarrow \tau_i\ \mathsf{type}\}_{0 \leq i < n_{exp}}$ *and* $\{\Delta \vdash \tau_i\ \mathsf{type}\}_{0 \leq i < n_{exp}}$
9. *(**Typing 4**)* $\{\langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \mathsf{parseUExp}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow e_i \Leftarrow \tau_i\}_{0 \leq i < n_{exp}}$ *and* $\{\Delta\ \Gamma \vdash e_i : \tau_i\}_{0 \leq i < n_{exp}}$
10. *(**Capture Avoidance**)* $e = [\{\tau'_i / t_i\}_{0 \leq i < n_{ty}}, \{e_i / x_i\}_{0 \leq i < n_{exp}}]e'$ *for some* $\{t_i\}_{0 \leq i < n_{ty}}$ *and* $\{x_i\}_{0 \leq i < n_{exp}}$ *and* e'
11. *(**Context Independence**)* $\mathsf{fv}(e') \subset \{t_i\}_{0 \leq i < n_{ty}} \cup \{x_i\}_{0 \leq i < n_{exp}}$

Proof. By rule induction over Rules (D.1). The proof follows the proof of Theorem B.31. □

Theorem D.13 (seTSM Abstract Reasoning Principles - Implicit Application)**.** *If*

$$\langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} /b/ \rightsquigarrow e \Leftarrow \tau$$

then:

1. *(**Typing 1**)* $\hat{\Psi} = \langle \mathcal{A}; \Psi, a \hookrightarrow \texttt{setsm}(\tau; e_{parse}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle$ *and* $\Delta\ \Gamma \vdash e : \tau$
2. $b \downarrow_{\mathsf{Body}} e_{body}$
3. $e_{parse}(e_{body}) \Downarrow \texttt{inj}[\mathtt{SuccessE}](e_{proto})$
4. $e_{proto} \uparrow_{\mathsf{PrExpr}} \grave{e}$
5. *(**Segmentation**)* $\mathsf{seg}(\grave{e})$ segments b
6. $\mathsf{summary}(\grave{e}) = \{\texttt{splicedt}[m'_i; n'_i]\}_{0 \leq i < n_{ty}} \cup \{\texttt{splicede}[m_i; n_i; \grave{\tau}_i]\}_{0 \leq i < n_{exp}}$
7. *(**Typing 2**)* $\{\langle \mathcal{D}; \Delta \rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow \tau'_i\ \mathsf{type}\}_{0 \leq i < n_{ty}}$ *and* $\{\Delta \vdash \tau'_i\ \mathsf{type}\}_{0 \leq i < n_{ty}}$
8. *(**Typing 3**)* $\{\emptyset \vdash^{\langle \mathcal{D}; \Delta \rangle; b} \grave{\tau}_i \rightsquigarrow \tau_i\ \mathsf{type}\}_{0 \leq i < n_{exp}}$ *and* $\{\Delta \vdash \tau_i\ \mathsf{type}\}_{0 \leq i < n_{exp}}$
9. *(**Typing 4**)* $\{\langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \mathsf{parseUExp}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow e_i \Leftarrow \tau_i\}_{0 \leq i < n_{exp}}$ *and* $\{\Delta\ \Gamma \vdash e_i : \tau_i\}_{0 \leq i < n_{exp}}$
10. *(**Capture Avoidance**)* $e = [\{\tau'_i / t_i\}_{0 \leq i < n_{ty}}, \{e_i / x_i\}_{0 \leq i < n_{exp}}]e'$ *for some* $\{t_i\}_{0 \leq i < n_{ty}}$ *and* $\{x_i\}_{0 \leq i < n_{exp}}$ *and* e'
11. *(**Context Independence**)* $\mathsf{fv}(e') \subset \{t_i\}_{0 \leq i < n_{ty}} \cup \{x_i\}_{0 \leq i < n_{exp}}$

Proof. By rule induction over Rules (D.2). The proof follows the proof of Theorem B.31, differing only in how the TSM definition is looked up. □

Lemma D.14 (Proto-Pattern Expansion Decomposition)**.** *If* $\grave{p} \rightsquigarrow p : \tau \dashv\!|^{\hat{\Delta}; \hat{\Phi}; b}\ \hat{\Gamma}$ *where*

$$\mathsf{summary}(\grave{p}) = \{\texttt{splicedt}[m'_i; n'_i]\}_{0 \leq i < n_{ty}} \cup \{\texttt{splicedp}[m_i; n_i; \grave{\tau}_i]\}_{0 \leq i < n_{pat}}$$

then all of the following hold:

1. $\{\hat{\Delta} \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow \tau'_i \ \mathsf{type}\}_{0 \le i < n_{ty}}$
2. $\{\emptyset \vdash^{\hat{\Delta}; b} \grave{\tau}_i \rightsquigarrow \tau_i \ \mathsf{type}\}_{0 \le i < n_{pat}}$
3. $\{\hat{\Delta} \vdash_{\hat{\Phi}} \mathsf{parseUPat}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow p_i : \tau_i \dashv\!\!| \ \hat{\Gamma}_i\}_{0 \le i < n_{pat}}$
4. $\hat{\Gamma} = \biguplus_{0 \le i < n_{pat}} \hat{\Gamma}_i$

Proof. By rule induction over Rules (D.8). The proof follows the proof of Theorem B.32. □

Theorem D.15 (spTSM Abstract Reasoning Principles - Explicit Application)**.** *If*

$$\hat{\Delta} \vdash_{\hat{\Phi}} \hat{a} \text{ `} b \text{` } \rightsquigarrow p : \tau \dashv\!\!| \ \hat{\Gamma}$$

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)*** $\hat{\Phi} = \hat{\Phi}', \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{sptsm}(\tau; e_{parse})$ *and* $\Delta \vdash p : \tau \dashv\!\!| \ \Gamma$
2. $b \downarrow_{\mathsf{Body}} e_{body}$
3. $e_{parse}(e_{body}) \Downarrow \mathtt{inj}[\mathtt{SuccessP}](e_{proto})$
4. $e_{proto} \uparrow_{\mathsf{PrPat}} \grave{p}$
5. ***(Segmentation)*** $\mathsf{seg}(\grave{p})$ $\mathsf{segments}$ b
6. $\mathsf{summary}(\grave{p}) = \{\mathtt{splicedt}[n'_i; m'_i]\}_{0 \le i < n_{ty}} \cup \{\mathtt{splicedp}[m_i; n_i; \grave{\tau}_i]\}_{0 \le i < n_{pat}}$
7. ***(Typing 2)*** $\{\hat{\Delta} \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow \tau'_i \ \mathsf{type}\}_{0 \le i < n_{ty}}$ *and* $\{\Delta \vdash \tau'_i \ \mathsf{type}\}_{0 \le i < n_{ty}}$
8. ***(Typing 3)*** $\{\emptyset \vdash^{\hat{\Delta}; b} \grave{\tau}_i \rightsquigarrow \tau_i \ \mathsf{type}\}_{0 \le i < n_{pat}}$ *and* $\{\Delta \vdash \tau_i \ \mathsf{type}\}_{0 \le i < n_{pat}}$
9. ***(Typing 4)*** $\{\hat{\Delta} \vdash_{\hat{\Phi}} \mathsf{parseUPat}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow p_i : \tau_i \dashv\!\!| \ \hat{\Gamma}_i\}_{0 \le i < n_{pat}}$
10. ***(No Hidden Bindings)*** $\hat{\Gamma} = \biguplus_{0 \le i < n_{pat}} \hat{\Gamma}_i$

Proof. By rule induction over Rules (D.4). The proof follows the proof of Theorem B.33. □

Theorem D.16 (spTSM Abstract Reasoning Principles - Implicit Application)**.** *If*

$$\hat{\Delta} \vdash_{\hat{\Phi}} /b/ \rightsquigarrow p : \tau \dashv\!\!| \ \hat{\Gamma}$$

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)*** $\hat{\Phi} = \langle \mathcal{A}; \Phi, a \hookrightarrow \mathtt{sptsm}(\tau; e_{parse}); \mathcal{I}, \tau \hookrightarrow a \rangle$ *and* $\Delta \vdash p : \tau \dashv\!\!| \ \Gamma$
2. $b \downarrow_{\mathsf{Body}} e_{body}$
3. $e_{parse}(e_{body}) \Downarrow \mathtt{inj}[\mathtt{SuccessP}](e_{proto})$
4. $e_{proto} \uparrow_{\mathsf{PrPat}} \grave{p}$
5. ***(Segmentation)*** $\mathsf{seg}(\grave{p})$ $\mathsf{segments}$ b
6. $\mathsf{summary}(\grave{p}) = \{\mathtt{splicedt}[n'_i; m'_i]\}_{0 \le i < n_{ty}} \cup \{\mathtt{splicedp}[m_i; n_i; \grave{\tau}_i]\}_{0 \le i < n_{pat}}$
7. ***(Typing 2)*** $\{\hat{\Delta} \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow \tau'_i \ \mathsf{type}\}_{0 \le i < n_{ty}}$ *and* $\{\Delta \vdash \tau'_i \ \mathsf{type}\}_{0 \le i < n_{ty}}$
8. ***(Typing 3)*** $\{\emptyset \vdash^{\hat{\Delta}; b} \grave{\tau}_i \rightsquigarrow \tau_i \ \mathsf{type}\}_{0 \le i < n_{pat}}$ *and* $\{\Delta \vdash \tau_i \ \mathsf{type}\}_{0 \le i < n_{pat}}$
9. ***(Typing 4)*** $\{\hat{\Delta} \vdash_{\hat{\Phi}} \mathsf{parseUPat}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow p_i : \tau_i \dashv\!\!| \ \hat{\Gamma}_i\}_{0 \le i < n_{pat}}$
10. ***(No Hidden Bindings)*** $\hat{\Gamma} = \biguplus_{0 \le i < n_{pat}} \hat{\Gamma}_i$

Proof. By rule induction over Rules (D.4). The proof follows the proof of Theorem B.33, differing only in how the parse function is looked up. □

Bibliography

[1] GHC Language Features - Syntactic extensions. https://downloads.haskell.org/~ghc/7.8.4/docs/html/users_guide/syntax-extns.html. 2.4.1

[2] Merlin, an assistant for editing OCaml code. https://the-lambda-church.github.io/merlin/. Retrieved Sep 6, 2016. 3.1.4

[3] Ocaml batteries included. http://batteries.forge.ocamlcore.org/. Retrieved May 31, 2016, . 1.1.2

[4] core - Jane Street Capital's standard library overlay. https://github.com/janestreet/core/. Retrieved May 31, 2016, . 1.1.2

[5] [Rust] Macros. https://doc.rust-lang.org/book/macros.html. Retrieved Nov. 3, 2015. 3.1.2

[6] SML/NJ Quote/Antiquote. http://www.smlnj.org/doc/quote.html. Retrieved Nov. 3, 2015. 2.4.2

[7] Information Technology Portable Operating System Interface (POSIX) Base Specifications, Issue 7. 2.4.1

[8] perlre - Perl regular expressions. http://perldoc.perl.org/perlre.html. Retrieved May 3, 2015. 1.1.2

[9] The Python Language Reference. http://docs.python.org. Retrieved June 4, 2014. 1.1.2

[10] OWASP Top 10 2013. https://www.owasp.org/index.php/Top_10_2013-Top_10. Retrieved Dec 12, 2013, 2013. 3

[11] Annika Aasa. Precedences for Conctypes. In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, 1993. URL http://doi.acm.org/10.1145/165180.165193. 2.4.4

[12] Annika Aasa. Precedences in Specifications and Implementations of Programming Languages. *Theor. Comput. Sci.*, 142(1):3–26, 1995. URL http://dx.doi.org/10.1016/0304-3975(95)90680-J. 2.4.4

[13] Annika Aasa, Kent Petersson, and Dan Synek. Concrete Syntax for Data Objects in Functional Languages. In *LISP and Functional Programming*, pages 96–105, 1988. URL http://doi.acm.org/10.1145/62678.62688. 2.4.4

[14] Peter Aczel. A general Church-Rosser theorem. Technical report, University of Manchester, 1978. 1.1

[15] Michael D. Adams. Towards the Essence of Hygiene. In *POPL*, 2015. URL http://doi.acm.org/10.1145/2676726.2677013. 2.4.10

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

[17] Andrew W. Appel and David B. MacQueen. Standard ML of new jersey. In *Programming Language Implementation and Logic Programming*, 1991. URL http://dx.doi.org/10.1007/3-540-54444-5_83. 2.1

[18] Alan Bawden. Quasiquotation in Lisp. In *Partial Evaluation and Semantic-Based Program Manipulation*, 1999. URL http://repository.readscheme.org/ftp/papers/pepm99/bawden.pdf. 6.2.1

[19] Matthias Blume. *The SML/NJ Compilation and Library Manager*, May 2002. Available from http://www.smlnj.org/doc/CM/index.html. 6.3

[20] George EP Box and Norman R Draper. *Empirical model-building and response surfaces.* John Wiley & Sons, 1987. 2.2

[21] Martin Bravenboer, Rob Vermaas, Jurgen Vinju, and Eelco Visser. Generalized type-based disambiguation of meta programs with concrete object syntax. In *Generative Programming and Component Engineering (GPCE)*, 2005. 2.4.8

[22] Martin Bravenboer, Eelco Dolstra, and Eelco Visser. Preventing Injection Attacks with Syntax Embeddings. In *GPCE*, 2007. URL http://doi.acm.org/10.1145/1289971.1289975. 3

[23] Eugene Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *4th Workshop on Scala*, 2013. 3, 2.4.10

[24] Stuart K. Card, Thomas P. Moran, and Allen Newell. The keystroke-level model for user performance time with interactice systems. *Commun. ACM*, 23(7):396–410, 1980. URL http://doi.acm.org/10.1145/358886.358895. 2.2

[25] Adam Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *PLDI*, 2010. URL http://doi.acm.org/10.1145/1806596.1806612. 1.2.1

[26] Adam Chlipala. Ur/Web: A simple model for programming the web. In *POPL*, 2015. URL http://dl.acm.org/citation.cfm?id=2676726. 1.2.1

[27] Adam Chlipala, Leaf Petersen, and Robert Harper. Strict bidirectional type checking. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI)*, 2005. URL http://doi.acm.org/10.1145/1040294.1040301. 7.2.3

[28] William D. Clinger and Jonathan Rees. Macros That Work. In *POPL*, 1991. URL http://doi.acm.org/10.1145/99583.99607. 2.4.10

[29] Russ Cox, Tom Bergan, Austin T. Clements, M. Frans Kaashoek, and Eddie Kohler. Xoc, an extension-oriented compiler for systems programming. In *ASPLOS*, 2008.

URL http://doi.acm.org/10.1145/1346281.1346312. 2.4.9

[30] Karl Crary. A syntactic account of singleton types via hereditary substitution. In *Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)*, 2009. URL http://doi.acm.org/10.1145/1577824.1577829. 8.2.5

[31] Ryan Culpepper, Scott Owens, and Matthew Flatt. Syntactic abstraction in component interfaces. In *International Conference on Generative Programming and Component Engineering (GPCE)*, 2005. 6.3

[32] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, 1982. 8.2.2

[33] Nils Anders Danielsson and Ulf Norell. Parsing Mixfix Operators. In *20th International Symposium on Implementation and Application of Functional Languages (IFL) - Revised Selected Papers*, 2008. URL http://dx.doi.org/10.1007/978-3-642-24452-0_5. 2.4.4, 2.4.4

[34] Rowan Davies. A Temporal-Logic Approach to Binding-Time Analysis. In *11th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 184–195, 1996. URL http://dx.doi.org/10.1109/LICS.1996.561317. 3

[35] F. DeRemer and H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, 2:80–86, 1976. 1.2.1

[36] Lukas Diekmann and Laurence Tratt. Eco: A language composition editor. In *International Conference on Software Language Engineering (SLE)*, 2014. 2.4.8

[37] Derek Dreyer. *Understanding and evolving the ML module system*. PhD thesis, Carnegie Mellon University, 2005. 2.3.2, 5.2.2

[38] Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. Modular type classes. In *POPL*, 2007. URL http://dl.acm.org/citation.cfm?id=1190216. 1.5

[39] Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*, 2013. URL http://dl.acm.org/citation.cfm?id=2500365. 1.4

[40] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1992. 2.4.10

[41] Sebastian Erdweg and Felix Rieger. A framework for extensible languages. In *GPCE*, 2013. 1.2.1, 2.4.5

[42] Sebastian Erdweg, Lennart C. L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. Growing a language environment with editor libraries. In *Generative Programming And Component Engineering (GPCE)*, 2011. URL http://doi.acm.org/10.1145/2047862.2047891. 2.4.8

[43] Sebastian Erdweg, Tillmann Rendel, Christian Kastner, and Klaus Ostermann. SugarJ: Library-based syntactic language extensibility. In *OOPSLA*, 2011. 1.2.1, 2.4.5

[44] Sebastian Erdweg, Felix Rieger, Tillmann Rendel, and Klaus Ostermann. Layout-sensitive language extensibility with SugarHaskell. In *Proceedings of the 2012 Symposium on Haskell*, pages 149–160. ACM, 2012. 2.4.5, 2.4.8

[45] Sebastian Erdweg, Tijs van der Storm, and Yi Dai. Capture-avoiding and hygienic program transformations. In *ECOOP*, 2014. URL `http://dx.doi.org/10.1007/978-3-662-44202-9`. 2.4.8

[46] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009. 2.4.5

[47] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP*, 2002. URL `http://doi.acm.org/10.1145/581478.581484`. 8.2.3

[48] Matthew Flatt. Creating languages in Racket. *Commun. ACM*, 55(1):48–56, January 2012. URL `http://doi.acm.org/10.1145/2063176.2063195`. 1.2.1, 2.4.6

[49] Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. Macros that Work Together - Compile-time bindings, partial expansion, and definition contexts. *J. Funct. Program.*, 22(2):181–216, 2012. URL `http://dx.doi.org/10.1017/S0956796812000093`. 1.2.2, 2.4.6

[50] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL*, 2004. URL `http://pdos.csail.mit.edu/~baford/packrat/popl04/peg-popl04.pdf`. 2.4.5, 2.4.8

[51] Bryan Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time. *CoRR*, abs/cs/0603077, 2006. URL `http://arxiv.org/abs/cs/0603077`. 2.4.5

[52] Murdoch J Gabbay and Andrew M Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3):341–363, 2002. 1.1

[53] Steven Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *ICFP*, 2001. 3, 2.4.10

[54] Carlo Ghezzi and Dino Mandrioli. Incremental parsing. *ACM Trans. Program. Lang. Syst.*, 1(1):58–70, January 1979. URL `http://doi.acm.org/10.1145/357062.357066`. 8.2.9

[55] T. R. G. Green. Cognitive dimensions of notations. In *Proceedings of the HCI'89 Conference on People and Computers V*, Cognitive Ergonomics, pages 443–460, 1989. 2.2

[56] T.R.G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7 (2):131–174, 1996. 2.2

[57] T.G. Griffin. Notational definition-a formal account. In *Logic in Computer Science (LICS '88)*, pages 372–383, 1988. 2.4.4

[58] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, March 1996. URL `http://doi.acm.org/10.1145/227699.227700`. 1.5, 8.2.3

[59] Robert Harper. Programming in Standard ML. http://www.cs.cmu.edu/~rwh/smlbook/book.pdf, 1997. Working draft, retrieved June 21, 2015. 1.1.2, 2.1

[60] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012. 4.2, 4.2.3

[61] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2nd edition, 2016. URL https://www.cs.cmu.edu/~rwh/plbook/2nded.pdf. 1.1, 2.1, 2.3.1, 3.1.2, 3.2.2, 5.2.2, 5.2.2, 8.2.3, A.1

[62] Robert Harper, John C Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *POPL*, 1989. 5.2.2

[63] T. P. Hart. MACRO definitions for LISP. Report A. I. MEMO 57, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, October 1963. 2.4.10

[64] Jan Heering, P. R. H. Hendriks, Paul Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989. URL http://doi.acm.org/10.1145/71605.71607. 2.4.5

[65] David Herman. *A Theory of Typed Hygienic Macros*. PhD thesis, Northeastern University, Boston, MA, May 2010. 2.4.10, 2.4.10

[66] David Herman and Mitchell Wand. A theory of hygienic macros. In *ESOP*, 2008. URL http://dx.doi.org/10.1007/978-3-540-78739-6_4. 2.4.10, 2.4.10, 3.2.11

[67] John E. Hopcroft. *Introduction to automata theory, languages, and computation*. Pearson Education India, 1979. 2.4.5

[68] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992. URL http://www.cs.nott.ac.uk/Department/Staff/gmh/parsing.ps. 2.4.6

[69] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *Journal of Functional Programming*, 8(04):437–444, 1998. 2.4.6

[70] Peter Zilahy Ingerman. "Pānini-Backus Form" Suggested. *Commun. ACM*, 10(3):137–, March 1967. URL http://doi.acm.org/10.1145/363162.363165. 2.4.5

[71] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, June 1993. 3

[72] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003. 1.1.2

[73] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, 2001. 2.4.9

[74] Antti-Juhani Kaijanaho. *Evidence-based programming language design: a philosophical and methodological exploration*. PhD thesis, University of Jyväskylä, 2015. 2.2

[75] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and ides. In *OOPSLA*, 2010. URL

http://doi.acm.org/10.1145/1869459.1869497. 2.4.5, 2.4.8

[76] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Symposium on LISP and Functional Programming*, pages 151–161, August 1986. 2.4.10

[77] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, AST-based API-usage analysis of open-source Java projects. In *ACM Symposium on Applied Computing (SAC)*, 2011. URL http://doi.acm.org/10.1145/1982185.1982471. 1.2.1

[78] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *POPL*, 2007. URL http://dl.acm.org/citation.cfm?id=1190216. 5.2.2

[79] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 4.02 Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, September 2014. 1.1.2, 1.2.1, 2, 2.4.5, 2.4.9

[80] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *ACM SIGPLAN Notices*, volume 9, pages 50–59. ACM, 1974. 2.3.2

[81] Florian Lorenzen and Sebastian Erdweg. Modular and automated type-soundness verification for language extensions. In *ICFP*, pages 331–342, 2013. URL http://dl.acm.org/citation.cfm?id=2500365. 2.4.5

[82] Florian Lorenzen and Sebastian Erdweg. Sound type-dependent syntactic language extension. In *POPL*, 2016. URL http://dl.acm.org/citation.cfm?id=2837614. 2.4.5, 2.4.8

[83] David MacQueen. Modules for Standard ML. In *Symposium on LISP and Functional Programming*, 1984. URL http://doi.acm.org/10.1145/800055.802036. 2.3.2

[84] Geoffrey Mainland. Why it's nice to be quoted: quasiquoting for Haskell. In *Haskell Workshop*, 2007. 2.4.10

[85] Michel Mauny and Daniel de Rauglaudre. A complete and realistic implementation of quotations for ML. In *Workshop on ML and its applications*, 1994. 2.4.10

[86] Coq Development Team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL http://coq.inria.fr. Version 8.0. 2.4.4, 2.4.4, 2.4.8, 8.2.3

[87] J. McCarthy. History of lisp. In *History of programming languages I*, pages 173–185. ACM, 1978. 3

[88] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004. 8.2.8

[89] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997. 1.1.2, 2.3.2, 6.3

[90] Stephan Albert Missura. *Higher-Order Mixfix Syntax for Representing Mathematical Notation and its Parsing*. PhD thesis, ETH Zurich, 1997. 2.4.4

[91] Peter Naur, John W Backus, Friedrich L Bauer, Julien Green, Charles Katz, John

McCarthy, Alan J Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, et al. Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1):1–17, 1963. 2.4.5

[92] Catarina Dutilh Novaes. *Formal languages in logic: A philosophical and cognitive analysis*. Cambridge University Press, 2012. 2.2

[93] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Inc., 2008. 1.1.2, 6.3, 8.2.3

[94] Chris Okasaki. Even higher-order functions for parsing or why would anyone ever want to use a sixth-order function? *Journal of Functional Programming*, 8(2): 195–199, March 1998. 2.4.6

[95] Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers. Active Code Completion. In *International Conference on Software Engineering (ICSE)*, pages 859–869, 2012. URL `http://dl.acm.org/citation.cfm?id=2337223.2337324`. 2.4.1, 1

[96] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely composable type-specific languages. In *ECOOP*, 2014. (document), 3.1, 3.2.5, 8.2.6

[97] Cyrus Omar, Chenglong Wang, and Jonathan Aldrich. Composable and hygienic typed syntax macros. In *ACM Symposium on Applied Computing (SAC)*, 2015. 3.1

[98] J.F. Pane and B.A. Myers. Usability issues in the design of novice programming systems. Technical report, Carnegie Mellon University, Human-Computer Interaction Institute, CMU-HCII-96-101, 1996. 2.2

[99] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. 2.1, 3.2.2

[100] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, January 2000. URL `http://doi.acm.org/10.1145/345099.345100`. 7.2.3, 8.2.2

[101] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004. 3.2.4

[102] Jon Rafkind and Matthew Flatt. Honu: syntactic extension for algebraic notation through enforestation. In *Generative Programming and Component Engineering (GPCE)*, pages 122–131, 2012. URL `http://doi.acm.org/10.1145/2371401.2371420`. 2.4.6

[103] J. C. Reynolds. GEDANKEN - a simple typless language based on the principle of completeness and reference concept. *Comm. A.C.M.*, 13(5), May 1970. 2

[104] J C Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*. September 1983. 5

[105] Nico Ritschel and Sebastian Erdweg. Modular capture avoidance for program transformations. In *SLE*, 2015. URL `http://dl.acm.org/citation.cfm?id=2814251`. 2.4.8, 2.4.8

[106] August Schwerdfeger and Eric Van Wyk. Verifiable composition of deterministic grammars. In *PLDI*, 2009. URL `http://doi.acm.org/10.1145/1542476.1542499`. 1.2.1, 2.4.8, 2.4.8, 2.4.8

[107] Dana Scott. Lambda calculus: some models, some philosophy. *Studies in Logic and the Foundations of Mathematics*, 101:223–265, 1980. 8.2.3

[108] Denys Shabalin, Eugene Burmako, and Martin Odersky. Quasiquotes for Scala. Technical Report EPFL-REPORT-185242, 2013. 1.1.2, 6.2.1

[109] Tim Sheard. Using MetaML: A staged programming language. *Lecture Notes in Computer Science*, 1608, 1999. 3, 2.4.10

[110] Tim Sheard and Nathan Linger. Programming in Omega. In *Central European Functional Programming School, Second Summer School (CEFP), Romania, June 23-30, 2007, Revised Selected Lectures*, pages 158–227, 2007. doi: 10.1007/978-3-540-88059-2_5. URL `http://dx.doi.org/10.1007/978-3-540-88059-2_5`. 2.4.1

[111] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Haskell Workshop*, 2002. 2.4.10, 2.4.10

[112] Erik Silkensen and Jeremy G. Siek. Well-typed islands parse faster. In *Trends in Functional Programming*, 2012. URL `http://dx.doi.org/10.1007/978-3-642-40447-4_5`. 2.4.8

[113] Konrad Slind. Object language embedding in Standard ML of New-Jersey. In *ICFP*, 1991. 2.4.2

[114] SML/NJ Documentation. The Visible Compiler. `http://www.smlnj.org/doc/Compiler/pages/compiler.html`. 3.1.2

[115] Eric Spishak, Werner Dietl, and Michael D Ernst. A type system for regular expressions. In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2012. 4

[116] Andreas Stefik and Susanna Siebert. An empirical investigation into programming language syntax. *TOCE*, 13(4):19, 2013. URL `http://doi.acm.org/10.1145/2534973`. 2.2

[117] Christopher A Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic (TOCL)*, 7(4):676–722, 2006. 5.2.2

[118] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F# 3.0*. Apress, 2012. 1.1.2

[119] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. 6.1

[120] Walid Taha and Patricia Johann. Staged notational definitions. In *GPCE*, 2003. URL `http://dx.doi.org/10.1007/978-3-540-39815-8_6`. 2.4.4

[121] D. R. Tarditi and A. W. Appel. *ML-Yacc, Version 2.0 Documentation*, 1990. 2.4.5

[122] Tim Teitelbaum and Thomas Reps. The Cornell Program Synthesizer: A Syntax-directed Programming Environment. *Commun. ACM*, 24(9):563–573, 1981. 2.4.8

[123] Ken Thompson. Programming techniques: Regular expression search algorithm.

Commun. ACM, 11(6):419–422, June 1968. URL http://doi.acm.org/10.1145/363347.363387. 2.3.2

[124] Arie van Deursen, Paul Klint, and Frank Tip. Origin tracking. *J. Symb. Comput.*, pages 523–545, 1993. URL http://dx.doi.org/10.1016/S0747-7171(06)80004-0. 3.1.2

[125] E. Visser. Stratego: A language for program transformation based on rewriting strategies (system description). In *Rewriting Techniques and Applications*, 2001. 2.4.5

[126] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards User-Friendly Projectional Editors. In *International Conference on Software Language Engineering (SLE)*, 2014. 2.4.8

[127] W3C. XHTML 1.0 The Extensible HyperText Markup Language. 1, 2000. 1.2.1

[128] Martin P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15 (4):147–161, 1994. 1.2.1

[129] Arthur Whitney and Dennis Shasha. Lots o'ticks: Real time high performance time series queries on billions of trades and quotes. *SIGMOD Rec.*, 30(2):617–617, May 2001. ISSN 0163-5808. doi: 10.1145/376284.375783. URL http://doi.acm.org/10.1145/376284.375783. 1.2.1

[130] Jacob Wieland. *Parsing Mixfix Expressions*. PhD thesis, Technische Universitat Berlin, 2009. 2.4.4

[131] Eric Van Wyk and August Schwerdfeger. Context-aware scanning for parsing extensible languages. In *GPCE*, 2007. URL http://doi.acm.org/10.1145/1289971.1289983. 1.2.1, 1.2.1, 2.4.5, 2.4.8

[132] Jeremy Yallop and Leo White. Modular macros (extended abstract). In *OCaml Users and Developers Workshop*, 2015. 6.3