Reasonably Programmable Syntax

Cyrus Omar

August 4, 2016

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, pattern matching, bidirectional typing, macro systems, hygiene, metaprogramming, domain-specific languages

TODO: dedication

Abstract

Language designers commonly define "syntactic sugar" for selected standard library constructs. For example, the designers of Standard ML (SML) defined derived forms for constructing and pattern matching on lists. Third-party library providers are, justifiably, envious of this special arrangement – they would like to be able to define syntactic sugar for constructing and pattern matching on values of the types that they have defined.

One approach is for these library providers to define "library-specific" syntax dialects. For example, the providers of a "collections" library might define a syntax dialect with derived forms for finite sets and dictionaries. The providers of a "web programming" library might define a syntax dialect with derived forms for HTML and JSON values. The problem is that library clients cannot combine dialects like these in a manner that conserves syntactic determinism, i.e. synactic conflicts can and do arise. Moreover, it can become difficult for library clients to reason in a disciplined manner about types and binding when examining the text of a program that uses unfamiliar derived forms, i.e. there are no clear principles of *syntactic abstraction*.

In this work, we introduce and formally define *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 (only 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 library clients to maintain:

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

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

Acknowledgments

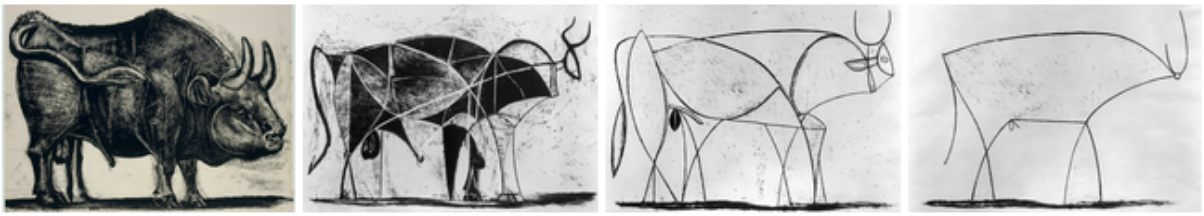
TODO: acknowledgments

Contents

List of Figures

Chapter 1

Introduction



Pablo Picasso (1881-1973)

1.1 Motivation

Experienced mathematicians and programmers define formal structures *compositionally*, drawing from libraries of other, often more abstract, formal structures. The benefits of this approach are clear: it saves time and improves reliability. The problem that motivates this work is that abstraction and composition can incur a non-trivial syntactic cost.

Consider, for example, natural numbers. It is straightforward to define the natural numbers as inductive structures – under the ambient mathematics of a document like this, we can do so by giving the following inductive grammar:

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

By defining natural numbers in this way, we immediately inherit a *structural induction principle*, i.e. to prove that some property P holds of all natural numbers, it suffices to establish $P(\mathbf{z})$ and $P(\mathbf{s}(n))$ assuming $P(n)$. The problem is that drawing particular natural numbers can become syntactically unwieldy (in fact, the syntactic cost of the drawing grows linearly with n.)[1]

Similarly, consider lists of natural numbers. Again, it is straightforward to define these as inductive structures:

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

[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 property of the structure itself.

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 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.

The problem again is that drawings of particular lists do not particularly resemble "naturally occurring" drawings of lists of numbers (e.g. on paper.)

Consider a third more sophisticated example (of particular relevance within this work): when defining or implementing a programming language, one often needs various sorts of ordered tree structures equipped with metaoperations[2] related to variable binding, e.g. bound variable renaming and capture-avoiding substitution. Repeatedly defining these structures "from scratch" is quite tedious, so language designers have instead developed a more general structure for working with *abstract binding trees (ABTs)* [11, 47]. Briefly, ABTs are ordered tree structures, classified into *sorts*, where each node is either a *variable*, x, or an *operation* of the following form:

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

Here, 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 renaming and substitution, and reasoning principles, like structural induction, that are necessary to define **Calc**'s semantics and prove it correct. Harper gives a complete account of ABTs, and many other examples of their use, in his textbook [47].

The problem with this approach is again that drawing non-trivial **Calc** expressions in operational form is syntactically costly:

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

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

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.

For example, in mathematics we generally use the Western Arabic numeral forms when working with 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. $[0, 1, 2]$ is taken 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 – an alternative *stylized form*:

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

and an alternative *textual form*:

```
1 / 2^(1/2)
```
(1.1c)

We can supplement these alternative primitive forms 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:

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

The reader can (when it is useful) 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) according to Figure 1.1:

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

When defining the semantics of a language like **Calc**, we 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]

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

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

Calc is semantically limited - we can express only simple arithmetic computations - so we should expect to need only a few more derived arithmetic forms to satisfyingly capture the idioms that arise in the limited domains where **Calc** might be useful.

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" languages. Consequently, the problems of syntactic cost just discussed 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 language.

We can continue to rely on the informal syntactic 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. program editors, interpreters and compilers. This, of course, is the regime of contemporary computer programming.

Constructing a formal syntax definition is not an especially difficult task for an experienced programmer, and there are many *syntax definition systems* that help with this task (we review these in Sec. 2.4.) The problem is that when designing a syntax for 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 might 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 [45, 71]. In SML, the derived expression form `[x, y, z]` desugars to an expression equivalent to:[4]

```
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".) Other languages similarly privilege select standard library constructs with derived forms:

- OCaml defines derived forms for strings (which are defined as arrays of characters.)
- Haskell defines derived forms for encapsulated commands (and, more generally, values of any type equipped with monadic structure.)
- Scala defines derived XML forms as well as string splicing forms, which capture the idioms of string concatenation.

[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.

- F#, Scala and various other languages define derived forms for encodings of the language's own terms (these are referred to as *quasiquotation* forms [85].)
- Python defines derived forms for mutable sets and dictionaries.
- Perl 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 from those defined in third-party libraries. In fact, the OCaml community has moved to de-emphasize the standard library in favor of competing bundles of third-party libraries (e.g. Batteries Included [2] is an open source effort, and Core [3] is a commercially maintained effort.)

1.2 Mechanisms of Syntactic Control

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

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

1.2.1 Syntax Dialects

When a library provider needs more syntactic control, one common approach 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 [19, 20]. Figure 1.2 demonstrates how XHTML expressions that contain strings can be drawn in Ur/Web. The desugaring of this form (not shown) is substantially more verbose and, for programmers familiar with the standardized syntax for XHTML, 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 [63], Copper [103] and SugarJ/Sugar* [31, 32], 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 this proliferation of syntax dialects is harmless or even desirable, because programmers can simply choose the right dialect for the job at hand [101].

However, this "dialect-oriented approach" is difficult to reconcile with the best practices of "programming in the large" [25], 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 the derived forms that they define together. This is problematic because client programs cannot be expected to fall cleanly into preconceived "problem domains" – large programs use a variety of libraries.

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 [37]), 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 equivalently 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 have no more than 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

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

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

Schwerdfeger and Van Wyk have developed a modular grammar analysis, implemented in Copper [103], that "nearly" guarantees that determinism is conserved when syntax dialects (of a certain restricted class) are combined [84], the caveat being that the constituent dialects must prefix all newly introduced forms with starting tokens drawn from disjoint sets. We will return to this requirement (and some other requirements of this approach) 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 other dialects of Standard ML's textual syntax.

```
1 val a = get_a()
2 val w = get_w()
3 val x = read_data(a)
4 val y = {|(!R)@&{&/x!/:2_!x}'!R}|}
```

Figure 1.3: An example of a confusing program fragment.

If the programmer happens to be familiar with the intentionally terse syntax of the stack-based database query processing language K, then Line 4 might pose few difficulties. If the programmer does not recognize this syntax, however, there are no simple, definitive protocols for answering questions like:

1. Which constituent dialect defined the derived form that appears on Line 4?
2. Are the characters `x` and `R` inside this derived form parsed as "spliced" expressions `x` and `R` (of variable form), or parsed in some other way peculiar to this form?
3. If `x` is the spliced expression `x`, does it refer to the binding on the previous line? Or was that binding shadowed by an unseen binding in the desugaring of Line 4?
4. If `w` is renamed, could that possibly break the program, or change its meaning? In other words, does the desugaring assume that some variable identified as `w` is in scope (though `w` does not appear directly in the text of Line 4)?
5. 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 `read_data` 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 `read_data`:

1. The language's syntax definition determines that `read_data(a)` is an expression of function application form.
2. Similarly, `read_data` and `a` are definitively expressions of variable form.
3. The variable `a` can only refer to the binding of `a` on Line 1.
4. The variable `w` can be renamed without knowing anything about the values that `read_data` and `a` stand for.
5. The type of `x` can be determined to be `B` by determining that the type of `read_data` is `A -> B` for some `A` and `B`, and checking that `a` has type `A`. Nothing else needs to be known about the values that `read_data` and `a` stand for. In Reynolds' words [82]:

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

1.2.2 Term Rewriting Systems

An alternative approach is to leave the context-free syntax of the language fixed, and instead allow library providers to contextually repurpose existing 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 relevant to 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 *macro systems*, however, have made substantial progress toward addressing these problems. In particular:

1. Macro systems require the client to explicitly apply the intended rewriting (implemented by a macro) to the term that is to be rewritten.
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 type structure [70]. That said, some progress has also been made on this front with the design of *typed macro systems*, like Scala's macro system [17], where

annotations constrain the macro arguments and the generated expansions.

The main problem with this approach, then, is that it leaves library providers quite syntactically constrained – they must find creative ways to repurpose existing forms. These existing forms normally have other meanings, so this can be quite confusing [79].

1.3 Contributions

In this work, we introduce a system of **typed syntax macros (TSMs)** that gives library providers substantially more syntactic control than existing typed macro systems while maintaining the ability to reason abstractly about types and binding.

Client programmers apply TSMs to *generalized literal forms* (which have no TSM-independent meaning.) For example, in Figure 1.4 we apply a TSM named `$html` to a generalized literal form delimited by backticks. TSM names must be prefixed by `$` to clearly distinguish TSM application from function application.

```
$html `<p>Hello, {[join ($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.

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. 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.

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.

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 green), can always be automatically generated no matter how each applied TSM has been implemented.

Notice that both arguments to `join` are themselves of TSM application form – the TSMs named `$str` and `$strlist` are applied to the generalized literal forms are 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`.

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

Figure 1.5: The segmentation of the example from Figure 1.4.

In order to reason about types and binding, client programmers need only have knowledge of 1) the segmentation (e.g. by examining a figure like this presented by a code editor or pretty-printer) and 2) type annotations on the definitions of the applied TSMs. No other details about the applied TSMs or the generated expansions need to be revealed to the client programmer. In other words, TSMs come equipped with useful principles of syntactic abstraction. We will, of course, more precisely characterize these reasoning principles as we proceed.

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 example 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 parameterized types 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 allow the expansions that TSMs generate to refer to application site bindings in a controlled and reasonable manner.
2. 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. We also demonstrate support for partial parameter application in TSM abbreviations, which decreases the syntactic cost of this explicit parameter passing style. Figure 1.6 demonstrates both of these features.

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

Figure 1.6: The example from Figure 1.4, expressed using a parametric TSM.

In these first chapters, we assume that each TSM definition is self-contained, needing no access to libraries or 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.

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 1.7 (assuming `join` has type `string -> list(string) -> string`.)

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

Figure 1.7: 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 1.7 to Figure 1.2), while maintaining the abstract reasoning principles characteristic of our approach.

The examples that we give are written in a 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 and, notionally, orthogonal to TSMs (e.g. higher-rank polymorphism [29], 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 presented herein. As such, all examples written in VerseML should be understood to be informal motivating material for the subsequent formal material.

We conclude in Chapter 8 with a discussion of present limitations and 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 types and binding.

1.4 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

[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.

the example of languages that support operator overloading or *ad hoc* polymorphism using type classes [28, 44], which also have some potential for "abuse" or "overuse".) The majority of programmers should very rarely need to define a TSM on their own.

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) [81]

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* [45] (a working draft can be found online.) We discuss integrating TSMs into languages from other design traditions in Sec. 8.6.3.

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* [45] 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) [13].)

The formal systems that we will consider are defined within the metatheoretic framework of type theory. More specifically, we will assume that abstract syntax trees (ASTs), abstract binding trees (ABTs, which enrich ASTs 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*) [47], except as otherwise stated. Familiarity with other formal accounts of type systems, e.g. Pierce's *Types and Programming Languages* (*TAPL*) [80], should also suffice.

2.2 Cognitive Cost

> *Typical views on formal languages are based on the premise that they are, above all, mathematical objects: a greater level of precision and technical complexity in logical investigations arises because they are precisely defined mathematical objects. Yet it would seem that viewing formal languages exclusively from this point of view offers a very partial and limited explanation of the impact that their use (and uses of formalisms more generally elsewhere) actually has. 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* [74]

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 produce or examine 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] Instead, we must turn to frameworks that are situationally useful [14].

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 [18].

One can also analyze cognitive cost using disciplined qualitative methods. For example, Green's *Cognitive Dimensions of Notations* [41, 42] and Pane and Myers' *Usability Issues* [79] (both of which synthesized much of the earlier work in the area) are highly

[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.

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 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 [92].) Kaijanaho provides another review of evidence-based language design methodologies [59].

There is much that remains to be understood about cognitive cost, particularly when the subject is an experienced programmer. Many of the difficulties that we will confront in this work have to do with the fact that allowing programmers to add new derived forms unconstrained to a syntax definition can decrease cognitive cost "in the small", i.e. for programmers who understand all of the details of the newly introduced desugaring transformations, while increasing cognitive cost "in the large" because programmers have few clear modular reasoning principles that they can rely on when they encounter an unfamiliar form. Our aim is to control cognitive cost at all scales.

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. 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* [47].) 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`, 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 variants of `T` and `e` is a value of the corresponding type. the labeled sum type above specifies two *variants*:

1. One variant, 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 variant, labeled `Cons`, takes values of the *product type* `int * list(int)`, the values of which are tuples.

Taken together, we can define two example values of type `list(int)` as follows:

```
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 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.

```
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* [97]. 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 (which is based on early work by MacQueen [67].) In particular, we 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: Definition of 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)* [64].) 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 `unfold_norm` that produces the *normal unfolding* of a given regex, i.e. a value of type `R.t u` 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) : R.t u u => (* ... *)

  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 -> R.t u u` 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 next task is to define auxiliary constructs that help to decrease the syntactic cost of expressions and patterns involving these constructs (and others like these.)

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 [88].

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:

```
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 [76]. 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 such mistakes and could not quickly diagnose the problem and determine a workaround if they had not used a regex library recently [76].

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 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 [8].

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 [16].

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" [91].

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.)

Another approach is to perform an extralinguistic static analysis that attempts to discover malformed statically determined regexes wherever they appear [91].

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 [5, 90], 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 [63]), 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* [57]. For example, if we integrated Davies'

temporal logic based approach into our language [24], 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 [86] and MacroML [40], 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 issues, 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 first to `e1`, then to `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.)

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

[6] In fact, quotation syntax can be expressed using parametric TSMs, which are the topic of Chapter 5, though we will leave the details as an exercise for the reader.

```
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`.

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 infix 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 early system of notational definitions [43], later variations on this system with better theoretical properties [94], and the syntax definition system integrated into the Agda programming language [23], 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 [69] 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 this system, 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. As stated directly in the Coq manual [69]:

> *Mixing different symbolic notations in a 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 [23] (this is related to earlier work by Aasa where a complete precedence graph was necessary [9].) 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 [72, 102]. These only reduce the likelihood of a conflict – they do not eliminate the possibility entirely.

2.4.5 Grammar-Based Syntax Definition Systems

Many syntax definition systems are oriented around *formal grammars* [53]. 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 [56].

Context-free grammars (CFGs) were first used to define the textual syntax of a major programming language – Algol 60 – by Backus [73]. 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 input text.

Most of the systems that we will describe in this section and the next section operate as language-external *preprocessors*, transforming source text into text accepted by a language's original syntax definition or, in some cases, directly into program trees. Some compilers, e.g. the OCaml compiler [63], integrate preprocessing into the build system. Other systems use a layer of directives placed in the source text to control preprocessor invocation. For example, in Racket's reader macro system, the programmer can direct the lexer (called the "reader") to shift control to a given parser when a designated directive or token is seen [37]. A few systems are integrated directly into a language definition – we will point these out when we introduce them.

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 [95], 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) [63]. In these systems, the output is an ML value computed by ML functions that appear associated with each production in the grammar (these functions are referred to as the *semantic actions*.)

The *syntax definition formalism (SDF)* [50] is a syntactic formalism for describing CFGs. SDF is used by a number of syntax definition systems, e.g. the Spoofax "language workbench" [60]. These systems commonly use Stratego, a rule-based rewriting language, as the language that output logic is written in [99]. 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 [32]. SugarHaskell is a similar system based on Haskell [34] and Sugar* simplifies the task of defining similar extensions of other languages [31]. SoundExt and SugarFOmega add the requirement that new derived forms must come equipped with derived typing rules [65]. 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 [36].) SoundX generalizes this idea to other base languages, and adds the ability to define type-dependent rewritings [66]. 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 [103]. We will say more about Copper when we discuss modular reasoning about syntactic determinism below.

```
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 $\mathcal{V}_{\texttt{rx}}$

Some other syntax definition systems are instead oriented around *parsing expression grammars* (PEGs) [38]. 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 [39].

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*) [54]. 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 'char 'out parser = 'char list -> ('out * ('char list)) list
```

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 : 'c 't parser -> 'c 't parser -> 'c 't parser
```

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 [55]. Okasaki has described an alternative design that uses continuations to control cost [75].

2.4.7 Examples of Syntax Dialects

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.[7]

[7] Technically, $\mathcal{V}_{\texttt{rx}}$ is a dialect of the textual syntax of VerseML with the generalized literal forms omitted – see Chapter 3.

```
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.[8]

$\mathcal{V}_{\texttt{rx}}$ 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, i.e. it constructs a regex from a string, `name`, and another regex, `ssn`. 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 subexpressions that appear inside literal forms as *spliced subexpressions*.

To splice in an expression that is not of variable form, e.g. a function application, we must delimit it with parentheses:

```
/@(capitalize name): %(ssn)/
```

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 sub-patterns*.

[8] In SML, where datatypes are generative, 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 next. In practice, programmers often fail to maintain context independence when using this technique.

```
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 -> t u` for computing the normal unfolding of the given regex, which is a value of type `u(R.t)`. $\mathcal{V}_{\text{RX}}$ defines derived forms for patterns matching values of types in the type family `'a u`. 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(R.t)`, whereas the patterns in Figure 2.10 match values of type `rx`.

These two examples were written in different syntax dialects. However, in general, it would be useful to have derived forms for values of type `rx` available even when we are working with a value of a type `R.t`, 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 [38].)

In response to this problem, Schwerdfeger and Van Wyk have developed a "nearly" 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 [84]. By relying on a context-aware scanner (a feature of Copper [103]) 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. The analysis is "nearly" modular in that only a relatively simple "combine-time" check that the set of marking terminals is disjoint is necessary.

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 in [84] 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 base 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 [15]. This approach is inefficient, particularly when a large number of grammars have been composed. Aasa et al. developed a dialect of ML where new quotation forms could be associated with datatype constructors, and integrated parsing of these quotation forms into the type inference mechanism of the language [10]. The method of *type-oriented island parsing* also integrates parsing with typechecking to improve efficiency [89].

A more radical approach would be to use a *language composition editor* like Eco [26].

```
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.

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 [96], has various costs and benefits, summarized in [100]. In this work, our interest is in text-based syntax, but we consider structure editors as future work in Sec. 8.3.

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 difficulties that programmers might encounter in analagous situations. In each case, we will discuss exceptional systems where these difficulties are ameliorated or avoided entirely.

1. **Search:** It is not always straightforward to determine which constituent dialect is responsible for any particular derived form.

   The system implemented by Copper [84] 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.

2. **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 [60] and Sugarclipse for SugarJ [33], 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.)

3. **Shadowing:** 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. This 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 shadowing if a client **`open`**s the module into scope.

   In most cases, shadowing 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, shadowing is by design. For example, in (Sugar)Haskell, **`do`** notation for monadic values introduces a new binding construct [34]. 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 shadowing.

   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 such conflicts. However, this is awkward and most systems provide no guarantee that the dialect provider maintained this freshness discipline [35].

   To enforce a prohibition on shadowing, 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 [69], enforce a prohibition on shadowing 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 shadowing [35, 83].[9]

4. **Context Dependence:** If the desugaring of a derived form assumes that certain identifiers are bound at the use 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 is 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.

5. **Typing:** Finally, and perhaps most importantly, 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 [66] and SoundX [83] 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 construct a derivation using the derived typing rules introduced by all of the constituent dialects, then examine this derivation to answer questions about the type of the desugaring and the spliced terms within it.

   Even for relatively simple base languages, like System $\mathbf{F}_\omega$, understanding a typing

[9] These papers refer to this property as "capture avoidance". We use the term "shadowing" rather than "capture" because "capture" has several incompatible meanings in the literature.

derivation requires significantly more expertise than programmers usually need.[10] For languages like ML, the judgement forms are substantially more complex.

Systems that rely on types to disambiguate between parses also better support reasoning about types. In particular, clients can determine the type of the desugaring by determining the types of subexpressions (which is not always trivial, due to the problems of **Search** and **Shadowing** discussed above) and, in some cases, examining the grammar production governing the form in question.

Due to the problems enumerated above, and the problem of conservatively combining syntax dialects, we do not integrate a syntax definition system into VerseML.

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 are given an entire compilation unit (e.g. a file) and generate a new compilation unit. 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 *extension points* into its textual syntax [63]. 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 causes a preprocessor distributed with `Lwt`, a lightweight threading library, to rewrite 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 [58] and the `xoc` compiler for C [22] both support user-defined non-local rewritings.

These systems present several conceptual problems, many of which are directly analagous to those that syntax definition systems present:

[10] At CMU, we teach ML to all first-year students (in 15-150.) However, understanding a judgemental 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.)

1. **Conflict:** Different preprocessors may recognize the same markers or code patterns.
2. **Search:** It is not always clear which preprocessor handles each rewritten form.
3. **Non-Locality:** 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 sub-terms appear inside rewritten forms (particularly string literals.)
5. **Shadowing:** 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.)

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 the problems of **Conflict**, **Search** and **Non-Locality** described above because macros are applied explicitly and operate locally.

Naïve macro systems, like the earliest variants of the LISP macro system [49] and early work on compile-time quotation expanders in ML [68], 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 to define a simple 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**, **Shadowing**, **Context Dependence** or **Typing** (assuming a language with a non-trivial type structure.)

The problem of **Shadowing** was addressed by the design of Scheme's *hygienic macro system* [12, 61], which automatically alpha-renames identifiers bound in the expansion so that they do not shadow (a.k.a. *capture*) those that appear at the macro application site. Nearly all modern macro systems (e.g. Scala's macro system [17] and the Template Haskell system [87]) incorporate such a hygiene mechanism.

The problem of **Context Dependence** is typically confronted by allowing macro expansions to explicitly refer only to those bindings that precede the macro definition site. These references are preserved even if the identifiers involved have been shadowed at the macro application site [12, 21, 30]. 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 some expansion generated by a macro might refer to, so 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 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. Some macro systems for languages with non-trivial type structure, like Template Haskell [87], also do not support reasoning about types. That said, there have been some formal studies of *typed macros* [51, 52] where type annotations appear on the macro to constrain the type of the generated expansion. The Scala macro system is a notable example of a practical typed macro system [17]. We are not aware of a typed macro system that has been integrated into a language with an ML-style module system.

In the remainder of this work, we will develop a system of *typed syntax macros (TSMs)*. As with other hygienic macro systems, TSMs do not permit shadowing of bindings at the application site. TSMs go further to addresses the remaining problems discussed above:

1. TSM expansions can refer to surrounding bindings (whether at the definition site or the application site) only through explicit parameters. Support for partial parameter application in TSM abbreviations will decrease the syntactic cost of this approach.
2. TSMs can parse spliced subexpressions out of literal bodies without violating context independence.
3. A segmentation of the literal body can always be determined.
4. Type annotations permit abstract reasoning about types. We will integrate TSMs into a language with parameterized types and an ML-style module system.

Part I

Simple TSMs

Chapter 3

Simple Expression TSMs (seTSMs)

This chapter introduces *simple expression TSMs (seTSMs)*, which will serve to introduce the essential character of TSMs. Subsequent chapters will introduce additional expressive power and address the various impracticalities of the system of seTSMs described in this chapter.

This chapter is organized as follows:

- Sec. 3.1 gives a "tutorial-style" introduction to seTSMs in VerseML.
- Sec. 3.2 then formally defines a reduced dialect of VerseML called 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 Simple Expression TSMs By Example

3.1.1 TSM Application

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

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

The context-free syntax of VerseML defines several generalized literal forms, 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.)

Generalized literal forms are left unparsed according to the context-free syntax. 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. 3.1.4. If proto-expansion validation succeeds, the language generates the *final expansion* (or more concisely, simply the *expansion*) of the TSM application. The program operates as if the expansion had appeared in place of the TSM application.

```
'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 [77] and multipart forms [78], but for simplicity we leave these out of VerseML.

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, which we will return to below, enforces this notion of context independence. (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`**.

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 [4].

The parse function is a *static function* delegated responsibility over parsing literal bodies. Static functions, marked by the **`static`** keyword, are functions that are applied during the typed expansion process. For this reason, 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 body = string

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

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

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. The resulting type distinguishes between parse errors and successful parses.[2] Let us consider these 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=`$e_{\texttt{msg}}$`, loc=`$e_{\texttt{loc}}$`})`

where $e_{\texttt{msg}}$ is an error message and $e_{\texttt{loc}}$ is a value of type `loc`, defined in Figure 3.2, that designates a subsequence of the literal body as the location of the error [98]. This information is for use by VerseML compilers when reporting the error to the programmer (but it otherwise has no semantic significance.)

Successful Parses If parsing succeeds, the parse function returns

`inj[Success](`$e_{\texttt{proto}}$`)`

where $e_{\texttt{proto}}$ 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 forms (just as in a compiler, c.f. SML/NJ's Visible Compiler library [6].) 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 variants are `SplicedT` and `SplicedE` – we will discuss these next, in Sec. 3.1.3.

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 [47], with only minor modifications to our semantics. Indeed, when we formally define seTSMs in Sec. 3.2, we abstract over the particular encoding scheme.

[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. 8.3.)

[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 loc)

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 loc)
```

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

3.1.3 Splicing

As described thusfar, TSMs operate just like term-rewriting macros over string literals. As such, TSMs do not create the problems of **Conflict**, **Search** and **Non-Locality**, 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 can refer to it within the computed encoding of the proto-expansion using the `SplicedE` variant of `proto_expr`. This variant takes a value of type `loc` because the proto-expansion must refer to spliced expressions by their zero-indexed location relative to the start of the literal body. This prevents TSMs from "forging" a spliced expression (i.e. claiming that an expression is a spliced expression when it does not appear in the literal body.)

For example, the proto-expansion generated by `$rx` for the literal body on the second line above, if written in a hypothetical textual syntax for proto-expressions where references to spliced expressions are written **`spliced`**`<startIdx, endIndex>`, is:

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

(For clarity of exposition, we again use the regex value constructors in lieu of explicit **`fold`** and **`inj`** operators and draw string values in the usual manner.) Here, **`spliced`**`<1, 4>` refers to the spliced expression `name` by location and **`spliced`**`<8, 10>` refers to the spliced expression `ssn` by location.

The parse function can similarly extract *spliced types* from a literal body using the `SplicedT` variant of `proto_typ`.

The semantics checks that all of the locations of spliced expressions and types within a proto-expansion are 1) in bounds relative to the literal body; and 2) non-overlapping. The *segmentation* induced by a proto-expansion is the finite set of these locations. For example, the segmentation of the proto-expansion above is:

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

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 this 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/
```

3.1.4 Proto-Expansion Validation

Three potential problems described in Secs. 2.4.9-2.4.10 remain: **Shadowing**, **Context Dependence** and **Typing**. Addressing these problems is the purpose of the *proto-expansion validation* process.

Shadowing

Proto-expansion validation ensures that spliced terms have access *only* to the bindings that appear at the application site. In other words, TSM cannot introduce new bindings into spliced terms. For example, suppose that `$rx` generated proto-expansions of the following essential form (drawn as above):

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

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 binding of `tmp` in the expansion of the TSM application. This complicates reasoning abstractly about binding. As such, proto-expansion validation ensures that shadowing does not occur. This prohibition on shadowing can be straightforwardly enforced by implicitly alpha-varying the bindings in the proto-expansion as needed. For example, the expansion of the example above might take the following form:

```
let tmp = (* ... application site tmp ... *) in
let tmp' = (* ... expansion 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 mechanism that generates "fresh variables".

TSM clients can, in turn, reliably determine exactly which bindings are available for use within every spliced expression without examining the expansion that the spliced expression eventually appears within. In other words, there can be no "hidden bindings".

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. We will return to this example when we consider other possible points in this design space in Sec. 8.6.6.

Context Dependence

The prohibition on shadowing ensures that variables that appear in spliced terms do not refer to bindings that appear in the surrounding expansion. The proto-expansion validation process also ensures that variables that appear in the proto-expansion do not refer to bindings that appear at the definition or application site. In other words, expansions must be *context independent*. A minimal example of a TSM that generates context-dependent proto-expansions is below:

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

The proto-expansion 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 application site typing contexts where `x` is bound. This "hidden assumption" makes reasoning about binding and renaming especially difficult, so this proto-expansion is deemed invalid (even when `$bad1` is applied where `x` happens to be 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. 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, 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.)

Typing

Finally, proto-expansion validation maintains a reasonable *typing discipline* by 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 will be of type `rx`. This addresses the problem of **Typing** described in Secs. 2.4.9-2.4.10, i.e. determining the type of an expansion does not require examining the expansion directly. Instead, it requires only examining the type annotation on the TSM definition (much as determining the type of a function application requires examining only the type of the function.)

3.1.5 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, under an environment where the regex value constructors are available:

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

(Again, due to the prohibition on context dependent expansions, the final expansion actually involves explicit **`fold`** and **`inj`** operators.)

3.1.6 Scoping

A benefit of specifying TSMs as a language primitive, rather than relying on extralinguistic mechanisms to manipulate the context-free syntax of our language directly, is that TSMs follow standard scoping rules.

For example, we can define a TSM that is visible only to a single expression like this:

```
let x =
  syntax $rx at rx by (* ... *) end in
    (* $rx is in scope here *)
in (* $rx is no longer in scope *)
```

If the **`in`** clause is omitted, as it was in the previous examples, the scope of the TSM extends to the end of the enclosing declaration (e.g. the function or module declaration.) We will consider how TSM definitions are packaged into libraries in Sec. 8.3.1.

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

```
1 val a = get_a()
2 val w = get_w()
3 val x = read_data(a)
4 val y = $k {|(!R)@&{&/x!/:2_!x}'!R}|}
```

Figure 3.4: The example from Figure 1.3, written using a TSM.

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 many situations.

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".

To underline this point, consider the program fragment in Figure 3.4, which is based on the example involving the K query language from Sec. 1.3. The programmer need not be familiar with the syntax of K, or examine the expansion itself, to answer questions corresponding to those posed in Sec. 1.3. In particular, the programmer knows that:

1. The TSM named `$k` is responsible for parsing the body of the literal on Line 4.
2. The character `x` inside the literal body is parsed as a "spliced" expression, `x`, as indicated by our visualization of the segmentation. The other characters, e.g. `R`, are definitively not spliced expressions.
3. The spliced expression `x` definitively refers to the binding of `x` on the previous line. No other binding of `x` could have shadowed this binding, due to the prohibition on shadowing.
4. The TSM application on Line 4 must be context-independent, so it cannot have referred to `w`.
5. We need only look at the type annotation on `$k` to determine the type of `y`. For example, if that declaration takes the following form, we know definitively that `y` has type `kquery` (without examining the elided parse function):

   ```
   syntax $k at kquery by (* ... *) end
   ```

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

To make the intuitions developed in the previous section mathematically precise, we will now introduce a reduced dialect of VerseML called miniVerse$_{\mathbf{SE}}$ that supports seTSMs. The full definition 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. The rule numbers below link to the corresponding rules 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

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
			$\texttt{fold}\{t.\tau\}(e)$	fold
			$\texttt{unfold}(e)$	unfold
			$\texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L})$	labeled tuple
			$\texttt{prj}[\ell](e)$	projection
			$\texttt{inj}[L;\ell]\{\{i \hookrightarrow \tau_i\}_{i\in L}\}(e)$	injection
			$\texttt{case}[L]\{\tau\}(e;\{i \hookrightarrow x_i.e_i\}_{i\in L})$	case analysis

Figure 3.5: Syntax of the miniVerse$_{\mathbf{SE}}$ expanded language (XL)

The syntax chart in Figure 3.5 defines the syntax of *types*, τ, and *(expanded) expressions*, e. Metavariables x range over 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. 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* [47] (or another text on type systems, e.g. *TAPL* [80]) 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 $x \in \mathrm{dom}(\Gamma)$, where $\mathrm{dom}(\Gamma)$ 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 $x \notin \mathrm{dom}(\Gamma)$, for the extension of Γ with a mapping from x to $x : \tau$, and $\Gamma \cup \Gamma'$ when $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \emptyset$ for the typing context mapping each $x \in \mathrm{dom}(\Gamma) \cup \mathrm{dom}(\Gamma')$ to $x : \tau$ if $x : \tau \in \Gamma$ or $x : \tau \in \Gamma'$.

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

3.2.4 Structural Dynamics

The *structural dynamics 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.7 (Iterated Transition)**.** *Iterated transition, $e \mapsto^* e'$, is the reflexive, transitive closure of the transition judgement, $e \mapsto e'$.*

Definition B.8 (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* (i.e. *by-value*) formulation of the dynamics.

Condition B.9 (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 $e = \texttt{fold}\{t.\tau'\}(e')$ 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* $e = \texttt{inj}[L', \ell; \ell]\{\{i \hookrightarrow \tau_i\}_{i \in L'}; \ell \hookrightarrow \tau'\}(e')$ *and* $\vdash e' : \tau'$ *and* e' val.

The Preservation condition ensures that evaluation preserve typing.

Condition B.10 (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.11 (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

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
			$\lambda \hat{x}{:}\hat{\tau}.\hat{e}$	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
			$\texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{by}\ \texttt{static}\ e\ \texttt{end}\ \texttt{in}\ \hat{e}$	seTSM definition
			$\hat{a}\ /b/$	seTSM application

Figure 3.6: Syntax of the miniVerse$_{\mathbf{SE}}$ unexpanded language (UL).

A 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 3.6 defines the relevant forms.

Unexpanded types and expressions are **not** abstract binding trees – we do **not** define notions of renaming, alpha-equivalence or substitution (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 mention variables at all, but rather *type identifiers*, $\hat{t}$, and *expression identifiers*, $\hat{x}$. Identifiers will be given meaning by expansion to variables during typed expansion. The distinction between identifiers and variables will be technically crucial.

All of the unexpanded forms in Figure 3.6 except the two TSM-related forms highlighted in yellow map onto expanded forms. We refer to these as the *common forms*. The mapping is given explicitly in Appendix B.2.1.

In addition to the stylized syntax given in Figure 3.6, 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. Our paper on Wyvern defines a textual syntax for a similar system [77]. 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.12 (Textual Representability)**.** *Both of the following must hold:*

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* $\mathsf{parseUExp}(b) = \hat{e}$.

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

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

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

$$\frac{}{\hat{\Delta}, \hat{t} \rightsquigarrow t\ \mathsf{type} \vdash \hat{t} \rightsquigarrow t\ \mathsf{type}} \quad \text{(B.5a)}$$

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

$$\frac{\hat{\Delta}, \hat{t} \rightsquigarrow t \text{ type} \vdash \hat{\tau} \rightsquigarrow \tau \text{ type}}{\hat{\Delta} \vdash \forall \hat{t}.\hat{\tau} \rightsquigarrow \texttt{all}(t.\tau) \text{ type}} \tag{B.5c}$$

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

$$\dfrac{\dfrac{\dfrac{}{\langle \hat{t} \rightsquigarrow t_2; t_1 \text{ type}, t_2 \text{ type} \rangle \vdash \hat{t} \rightsquigarrow t_2 \text{ type}} \text{(B.5a)}}{\langle \hat{t} \rightsquigarrow t_1; t_1 \text{ type} \rangle \vdash \forall \hat{t}.\hat{t} \rightsquigarrow \texttt{all}(t_2.t_2) \text{ type}} \text{(B.5c)}}{\langle \varnothing; \varnothing \rangle \vdash \forall \hat{t}.\forall \hat{t}.\hat{t} \rightsquigarrow \texttt{all}(t_1.\texttt{all}(t_2.t_2)) \text{ type}} \text{(B.5c)}$$

Notice that whenever a type identifier is bound, 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 fresh hypothesis. Without this mechanism, expansions for unexpanded types where an inner binding shadows an outer binding, 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.26 (Type Expansion). *If* $\langle \mathcal{D}; \Delta \rangle \vdash \hat{\tau} \rightsquigarrow \tau \text{ type}$ *then* $\Delta \vdash \tau \text{ type}$.

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 $\hat{x} \in \text{dom}(\mathcal{G})$ 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

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

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.30 (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.6k) handle unexpanded expressions of common form. The first five of these rules are reproduced below:

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

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

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

$$\frac{\hat{\Delta}, \hat{t} \rightsquigarrow t\ \mathsf{type}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \Lambda \hat{t}.\hat{e} \rightsquigarrow \mathtt{tlam(}t.e\mathtt{)} : \mathtt{all(}t.\tau\mathtt{)}} \tag{B.6d}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \mathtt{all(}t.\tau\mathtt{)} \qquad \hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'\ \mathsf{type}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e}\mathtt{[}\hat{\tau}'\mathtt{]} \rightsquigarrow \mathtt{tap\{}\tau'\mathtt{\}(}e\mathtt{)} : [\tau'/t]\tau} \tag{B.6e}$$

These rules are entirely straightforward, mirroring the corresponding typing rules, i.e. Rules (B.2). In particular, observe that, in each of these rules, the unexpanded and expanded expression forms in the conclusion correspond, and each premise corresponds to a premise of the corresponding typing rule. The type assigned in the conclusion of each rule above 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.30 are by straightforward induction and application of the corresponding typing rule.

seTSM Definition and Application The two remaining typed expansion rules, Rules (B.6l) and (B.6m), 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

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

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.6l) defines typed expansion of this form:

$$\frac{\begin{array}{c}\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \varnothing\,\varnothing \vdash e_{\mathrm{parse}} : \mathtt{parr(Body;ParseResultSE)} \\ \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{setsm(}\tau;e_{\mathrm{parse}}\mathtt{)}} \hat{e} \rightsquigarrow e : \tau'\end{array}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{by}\ \mathtt{static}\ e_{\mathrm{parse}}\ \mathtt{end}\ \mathtt{in}\ \hat{e} \rightsquigarrow e : \tau'} \tag{B.6l}$$

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.21 (Body Isomorphism)**.** *All of the following must hold:*

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

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

`ParseResultSE` abbreviates a labeled sum type that distinguishes parse errors from successful parses:[4]

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

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

The type abbreviated `ProtoExpr` classifies encodings of *proto-expressions*, *è* (pronounced "grave *e*".) The syntax of proto-expressions, defined in Figure 3.7, will be described when we describe proto-expansion validation in Sec. 3.2.9. The mapping from proto-expressions to values of type `ProtoExpr` is defined by the *proto-expression encoding judgement*, $\grave{e} \downarrow_{\mathsf{ProtoExpr}} e$. An inverse mapping is defined by the *proto-expression decoding judgement*, $e \uparrow_{\mathsf{ProtoExpr}} \grave{e}$.

[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.

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

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

Condition B.23 (Proto-Expression Isomorphism)**.** *All of the following must hold:*

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

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

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

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

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

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

3. The final premise of Rule (B.6l) 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.6l) 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 $\hat{a} \in \mathrm{dom}(\mathcal{A})$ 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 $a \in \mathrm{dom}(\Psi)$ to an *expanded seTSM definition*, $a \hookrightarrow \mathtt{setsm}(\tau; e_{\mathrm{parse}})$, where τ is the seTSM's type annotation, and e_{parse} is its parse function. We write $\Psi, a \hookrightarrow \mathtt{setsm}(\tau; e_{\mathrm{parse}})$ when $a \notin \mathrm{dom}(\Psi)$ for the extension of Ψ that maps a to $a \hookrightarrow \mathtt{setsm}(\tau; e_{\mathrm{parse}})$.

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

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

   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:

$$\hat{a}\ /b/$$

This stylized form uses forward slashes 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:

$$\frac{\begin{array}{c}\hat{\Psi} = \hat{\Psi}', \hat{a} \rightsquigarrow a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}}) \\ b \downarrow_{\text{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{SuccessE} \cdot e_{\text{proto}} \qquad e_{\text{proto}} \uparrow_{\text{ProtoExpr}} \grave{e} \\ \text{seg}(\grave{e})\ \text{segments}\ b \qquad \varnothing\,\varnothing \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};b} \grave{e} \rightsquigarrow e : \tau\end{array}}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi}} \hat{a}\ /b/ \rightsquigarrow e : \tau} \tag{B.6m}$$

where:

$$\texttt{SuccessE} \cdot e \stackrel{\text{def}}{=} \texttt{inj}[L_{\text{SE}}; \texttt{SuccessE}]\{\texttt{ParseError} \hookrightarrow \langle\rangle, \texttt{SuccessE} \hookrightarrow \texttt{ProtoExpr}\}(e)$$

The premises of Rule (B.6m) 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, i.e. a value of the form abbreviated $\texttt{SuccessE} \cdot e_{\text{proto}}$ (as shown) results from evaluation, then e_{proto} will be a value of type `ProtoExpr` (assuming a well-formed seTSM context, by application of the Preservation assumption, Assumption B.10.) 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, $\text{seg}(\grave{e})$ and ensures that it is valid with respect to b.

   A *segmentation*, ψ, is a finite set of *segments* of the form $\langle m; n; \text{UExp}\rangle$ or $\langle m; n; \text{UTyp}\rangle$. The metafunction $\text{seg}(\grave{e})$ determines the segmentation of $\grave{e}$ by generating one segment for each reference to a spliced expression or type, respectively (see next section.)

   The predicate ψ 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.
6. The final premise of Rule (B.6m) *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.

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
		$\texttt{splicedt}[m;n]$	$\texttt{splicedt}\langle m,n\rangle$	spliced
PrExp	$\grave{e}$ ::=	x	x	variable
		$\texttt{prlam}\{\grave{\tau}\}(x.\grave{e})$	$\lambda x{:}\grave{\tau}.\grave{e}$	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
		$\texttt{prfold}\{t.\grave{\tau}\}(\grave{e})$	$\texttt{fold}(\grave{e})$	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
		$\texttt{prinj}[L;\ell]\{\{i \hookrightarrow \grave{\tau}_i\}_{i\in L}\}(\grave{e})$	$\texttt{inj}[\ell](\grave{e})$	injection
		$\texttt{prcase}[L]\{\tau\}(\grave{e};\{i \hookrightarrow x_i.\grave{e}_i\}_{i\in L})$	$\texttt{case}\ \grave{e}\ \{i \hookrightarrow x_i.\grave{e}_i\}_{i\in L}$	case analysis
		$\texttt{splicede}[m;n]$	$\texttt{splicede}\langle m,n\rangle$	spliced

Figure 3.7: Syntax of $\text{miniVerse}_{\mathbf{SE}}$ proto-types and proto-expressions.

3.2.9 Syntax of Proto-Expansions

Figure 3.7 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 3.7: 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*, $\texttt{splicede}[m;n]$, 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
$\Delta \vdash^{\mathbb{T}} \grave{\tau} \rightsquigarrow \tau$ type	$\grave{\tau}$ has well-formed expansion τ
$\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau$	$\grave{e}$ has expansion e of type τ

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 $\text{ts}(\mathbb{E})$ 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.6m) above). These structures will be necessary to validate the references to spliced unexpanded types and expressions that appear within a proto-expansion.

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).

Common Forms Rules (B.9a) through (B.9f) validate proto-types of common form. The first three of these are reproduced below.

$$\frac{}{\Delta, t\ \mathsf{type} \vdash^{\mathbb{T}} t \rightsquigarrow t\ \mathsf{type}} \tag{B.9a}$$

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

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

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.

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. Indeed, $\mathbb{T}$ passes opaquely through the rules above.

References to Spliced Types The only proto-type form that does not correspond to a type form is $\mathtt{splicedt}[m; n]$, 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, b, 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:

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

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 $\mathsf{parseUTyp}(b)$, 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, $\langle \mathcal{D}; \Delta_{\text{app}} \rangle$, 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 shadowing of type variables as described in Sec. 3.1.4 – the TSM provider can choose type variable names freely within a proto-expansion.

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.27 (Proto-Expansion 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* $\Delta \cup \Delta_{app} \vdash \tau\ \mathsf{type}$.

Proto-Expression Validation

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

Common Forms Rules (B.10a) through (B.10k) validate proto-expressions of common form. The first three of these rules are reproduced below:

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

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

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

Once again, these rules mirror the typing rules, i.e. Rules (B.2). The expression splicing scene, $\mathbb{E}$, passes opaquely through these rules.

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. Indeed, $\mathbb{E}$ is not inspected by any of the rules above. This achieves *context-independent expansion* as described in Sec. 3.1.4 – 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.

References to Spliced Unexpanded Expressions The only proto-expression form that does not correspond to an expanded expression form is $\texttt{splicede}[m;n]$, 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.10l) governs this form:

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

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 $\mathsf{parseUExp}(b)$, characterized in Sec. 3.2.5, to produce the referenced spliced unexpanded expression, $\hat{e}$.

The second premise of Rule (B.10l) 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 $\hat{e}$.

The third premise of Rule (B.10l) 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 fourth 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 three premises enforce the prohibition on shadowing as described in Sec. 3.1.4 – the TSM provider can choose variable names freely within a proto-expansion, because the language prevents them from shadowing those at the application site.

3.2.11 Metatheory

Typed Expansion

To prove Theorem B.30, which was mentioned at the beginnning of Sec. 3.2.6, we must prove the following stronger theorem (because the proto-expression validation judgement is defined mutually inductively with the typed expansion judgement):

Theorem B.29 (Typed Expansion (Full))**.** *Both of the following hold:*

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.2.3. 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.6l)**.** We have

(1) $\hat{e} = \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}'\ \texttt{by}\ \texttt{static}\ e_{\text{parse}}\ \texttt{end}\ \texttt{in}\ \hat{e}'$ by assumption

(2) $\hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'$ 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) $\Delta \vdash \tau'$ type — by Lemma B.26 to (2)
(6) $\Delta\,\Gamma \vdash e : \tau$ — by IH, part 1(a) on (4)

Case (B.6m). We have

(1) $\hat{e} = \hat{a}\ /b/$ — 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) $b \downarrow_{\text{Body}} e_{\text{body}}$ — by assumption
(5) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{SuccessE} \cdot e_{\text{proto}}$ — by assumption
(6) $e_{\text{proto}} \uparrow_{\text{ProtoExpr}} \grave{e}$ — 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) $\emptyset \cap \text{dom}(\Gamma) = \emptyset$ — 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.10l).

(1) $\grave{e} = \texttt{splicede}[m; n]$ — by assumption
(2) $\text{parseUExp}(\text{subseq}(b; m; n)) = \hat{e}$ — by assumption
(3) $\hat{\Delta}_{\text{app}}\,\hat{\Gamma}_{\text{app}} \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau$ — by assumption
(4) $\Delta \cap \Delta_{\text{app}} = \emptyset$ — by assumption
(5) $\text{dom}(\Gamma) \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset$ — by assumption
(6) $\Delta_{\text{app}}\,\Gamma_{\text{app}} \vdash e : \tau$ — by IH, part 1 on (3)
(7) $\Delta \cup \Delta_{\text{app}}\,\Gamma \cup \Gamma_{\text{app}} \vdash e : \tau$ — by Lemma B.2 over Δ and Γ and exchange on (6)

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.6m). There, we have that the metric remains stable:

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

The only case in the proof of part 2 that invokes part 1 is Case (B.10l). 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:

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

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.13 (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.22 (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. □

We can now restate Theorem B.30.

Theorem B.30 (Typed Expression Expansion). *If* $\langle \mathcal{D}; \Delta \rangle \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau$ *then* $\Delta\ \Gamma \vdash e : \tau$.

Proof. This theorem follows immediately from Theorem B.29, part 1. □

Reasoning Principles

The following theorem, together with Theorem B.30, establishes **Typing**, **Segmentation** and **Context Independence** as discussed in Sec. 3.1.4.

Theorem B.31 (seTSM Typing and Context Independence). *If* $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{a}\ /b/ \rightsquigarrow e : \tau$ *then:*

1. ***(Typing)*** $\hat{\Psi} = \hat{\Psi}', \hat{a} \rightsquigarrow a \hookrightarrow \texttt{setsm}(\tau; e_{parse})$
2. $b \downarrow_{\mathsf{Body}} e_{body}$
3. $e_{parse}(e_{body}) \Downarrow \texttt{SuccessE} \cdot e_{proto}$
4. $e_{proto} \uparrow_{\mathsf{ProtoExpr}} \grave{e}$
5. ***(Segmentation)*** $\mathsf{seg}(\grave{e})$ segments b
6. ***(Context Independence)*** $\emptyset\ \emptyset \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};b} \grave{e} \rightsquigarrow e : \tau$

Proof. By rule induction over Rules (B.6). The only rule that applies is Rule (B.6m). The conclusions of the theorem are the premises of this rule. □

The following theorem establishes a prohibition on **Shadowing** as discussed in Sec. 3.1.4.

Theorem B.32 (Shadowing Prohibition)**.**

1. *If* $\Delta \vdash^{\langle \mathcal{D}; \Delta_{app} \rangle; b} \texttt{splicedt}[m; n] \rightsquigarrow \tau$ type *then:*
   (a) $\mathsf{parseUTyp}(\mathsf{subseq}(b; m; n)) = \hat{\tau}$
   (b) $\langle \mathcal{D}; \Delta_{app} \rangle \vdash \hat{\tau} \rightsquigarrow \tau$ type
   (c) $\Delta \cap \Delta_{app} = \emptyset$
2. *If* $\Delta\ \Gamma \vdash^{\langle \mathcal{D}; \Delta_{app} \rangle; \langle \mathcal{G}; \Gamma_{app} \rangle; \hat{\Psi}; b} \texttt{splicede}[m; n] \rightsquigarrow e : \tau$ *then:*
   (a) $\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e}$
   (b) $\langle \mathcal{D}; \Delta_{app} \rangle\ \langle \mathcal{G}; \Gamma_{app} \rangle \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau$
   (c) $\Delta \cap \Delta_{app} = \emptyset$
   (d) $dom(\Gamma) \cap dom(\Gamma_{app}) = \emptyset$

Proof.

1. By rule induction over Rules (B.9). The only rule that applies is Rule (B.9g). The conclusions are the premises of tihs rule.
2. By rule induction over Rules (B.10). The only rule that applies is Rule (B.10l). The conclusions are the premises of tihs rule.

□

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, recall 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 already discussed in Sec. 2.3):

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

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

Variable patterns match any value. In the corresponding branch, the variable stands for that value. A variables can appear only once in a pattern. For example, on Line 3 above, the pattern

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

matches values with the following structure:

```
Seq(Str(e1), Seq(Str ": ", e2))
```

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, i.e. `Some (name, ssn)`.

On Line 4 above, the pattern _ is the *wildcard pattern* – it matches any value of the appropriate type and 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*. Most 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, most 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. $\text{miniVerse}_{\mathbf{SE}}$ from the previous section.)

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

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

Dialect formation is not a modular approach, for the reasons discussed in Chapter 2.4.8.

Expression TSMs – introduced in Chapter 3 – can decrease the syntactic cost of constructing a value of a recursive labeled sum type like `rx`. However, expressions are syntactically distinct from patterns, so we cannot simply apply an expression TSM to generate a pattern.[1] For this reason, we need to introduce 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 $\text{miniVerse}_{\mathbf{SE}}$ called $\text{miniVerse}_{\mathbf{S}}$ 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
```

[1] The fact that certain concrete expression and pattern forms overlap is immaterial to this fundamental distinction. There are many expression forms that the expansion generated by an expression TSM might use that have no corresponding pattern form, e.g. lambda abstraction.

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. 4.1.4. 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 in the usual manner.

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:

```
1  fun is_dna_rx(r : rx) : boolean =>
2    match r using $rx with
3    | /A/ => True
4    | /T/ => True
5    | /G/ => True
6    | /C/ => True
7    | /%(r1)%(r2)/ => (is_dna_rx r1) andalso (is_dna_rx r2)
8    | /%(r1)|%(r2)/ => (is_dna_rx r1) andalso (is_dna_rx r2)
9    | /%(r)*/ => is_dna_rx r'
10   | _ => False
11   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 = (* no variable pattern form *)
               + Wild
               + (* ... *)
               + SplicedP of loc
```

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

```
syntax $rx at rx for expressions by
  static fn(body : body) : proto_expr parse_result =>
    (* regex expression parser here *)
and 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. 4.1.4 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 `loc` 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.)

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>`, is:

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

Here, **spliced**`<1, 4>` refers to the subpattern `name` by position, and **spliced**`<8, 10>` refers to the subpattern `ssn` by position.

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

non-overlapping. The *segmentation* of the literal body is the finite set of spliced pattern locations, e.g. here the following finite set:

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

An editor or pretty-printer can convey this segmentation information using color, as shown in the examples above.

4.1.4 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 the proto-pattern against the type annotation specified by the pattern TSM, e.g. the type `rx` above.

Binding

To maintain a useful binding discipline, i.e. to allow programmers to reason about variable binding without examining TSM expansions directly, the validation process restricts variable patterns (e.g. `name` and `ssn` above): variable patterns can 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 shadowing discussed in Sec. 3.1.4 (differing in that it is concerned with the bindings visible to the corresponding branch expression, rather than to spliced expressions.)

In VerseML, patterns are context-independent by construction (i.e. there is no way to refer to the surrounding bindings from within a pattern), so there is no need to consider the question of context independence. However, in languages that support, e.g., arbitrary expressions as *guards* within patterns (e.g. OCaml [63]), or in languages that support pattern synonyms, it would be necessary also to enforce context independence, following the approach taken with seTSMs (see Sec. 3.1.4.)

4.1.5 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))
```

Sort			Operational Form	Description
Typ	τ	::=	$\cdots$	(see Figure 3.5)
Exp	e	::=	$\cdots$	(see Figure 3.5)
			$\texttt{match}[n]\{\tau\}(e; \{r_i\}_{1 \leq i \leq n})$	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

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

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. miniVerse$_\mathbf{S}$ consists of an *unexpanded language* (UL) defined by typed expansion to a standard *expanded language* (XL), like miniVerse$_\mathbf{SE}$. 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* [46].

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 minor technical note: the ABT *renaming* metaoperation (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 total ordering on labels, $\leq$.

[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.

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'
$\Delta \vdash p : \tau \dashv\!\shortmid \Gamma$	p matches values of type τ and generates hypotheses Γ

The types of $\text{miniVerse}_{\mathbf{S}}$ are exactly those of $\text{miniVerse}_{\mathbf{SE}}$, 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 $\text{miniVerse}_{\mathbf{SE}}$, i.e. Rules (B.2a) through (B.2k).
- The following rule for match expressions:

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

The first premise of Rule (B.2l) assigns a type, τ, to the scrutinee, e. The second premise checks that the type of the expression as a whole, τ', is well-formed.[3] The third premise then ensures that each rule r_i, for $1 \leq i \leq n$, 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:

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

The first premise invokes the *pattern typing judgement*, $\Delta \vdash p : \tau \dashv\!\shortmid \Gamma$, 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 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:

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

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

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

[3] The second premise of Rule (B.2l), and the type argument in the match form, are necessary to maintain regularity, defined below, but only because when $n = 0$, the type τ' is arbitrary. In all other cases, τ' can be determined by assigning types to the branch expressions.

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

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

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, $\cup_{i\in L}\Gamma_i$:

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

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

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

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:

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

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

4.2.3 Structural Dynamics

The *structural dynamics of* 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.7 (Iterated Transition)**.** *Iterated transition, $e \mapsto^* e'$, is the reflexive, transitive closure of the transition judgement, $e \mapsto e'$.*

Definition B.8 (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 miniVerse$_{\mathbf{SE}}$, i.e. Condition B.9.

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

Condition B.10 (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.11 (Progress). *If $\vdash e : \tau$ then either e* val *or e* 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 [46] for a formal account.)

4.2.4 Syntax of the Unexpanded Language

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

Sort			Stylized Form	Description
UTyp	$\hat{\tau}$	::=	$\cdots$	(see Figure 3.6)
UExp	$\hat{e}$	::=	$\cdots$	(see Figure 3.6)
			`match` $\hat{e}$ $\{\hat{r}_i\}_{1 \leq i \leq n}$	match
			`syntax` $\hat{a}$ `at` $\hat{\tau}$ `for patterns by static` e `end in` $\hat{e}$	spTSM definition
URule	$\hat{r}$	::=	$\hat{p} \Rightarrow \hat{e}$	match rule
UPat	$\hat{p}$	::=	$\hat{x}$	identifier pattern
			_	wildcard pattern
			`fold(`$\hat{p}$`)`	fold pattern
			$\langle \{i \hookrightarrow \hat{p}_i\}_{i \in L} \rangle$	labeled tuple pattern
			`inj`$[\ell](\hat{p})$	injection pattern
			$\hat{a}$ `/`b`/`	spTSM application

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

As in 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 also 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 3.6, 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.12 (Textual Representability)**.** *All of the following must hold:*

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

Type Expansion

Unexpanded type formation contexts, $\hat{\Delta}$, consist of a type identifier expansion context, $\mathcal{D}$, and a type formation context, Δ. These were defined in Sec. 3.2.6. The *type expansion judgement*, $\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type}$, is inductively defined by Rules (B.5) as before.

Typed Expression, Rule and Pattern Expansion

Unexpanded typing contexts, $\hat{\Gamma}$, consist of an expression identifier expansion context, $\mathcal{G}$, and a typing context, Γ. seTSM contexts, $\hat{\Psi}$, consist of a TSM identifier expansion context, $\mathcal{A}$, and an seTSM definition context, Ψ. These were all defined in Sec. 3.2.6. We will define *spTSM contexts*, $\hat{\Phi}$, below.

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

Rules (B.6a) through (B.6m) 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.6n) governs this form:

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Phi};\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau \qquad \Delta \vdash \tau'\ \mathsf{type} \qquad \{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \rightsquigarrow r_i : \tau \Mapsto \tau'\}_{1 \leq i \leq n}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{match}\ \hat{e}\ \{\hat{r}_i\}_{1 \leq i \leq n} \rightsquigarrow \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \leq i \leq n}) : \tau'} \quad \text{(B.6n)}$$

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

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

Because unexpanded terms mention only expression identifiers, which are given meaning by expansion to variables, the pattern typing rules must generate both a 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.

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

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

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

$$\frac{\{\Delta \vdash_{\hat{\Phi}} \hat{p}_i \rightsquigarrow p_i : \tau_i \dashv\Vdash \hat{\Gamma}_i\}_{i \in L}}{\Delta \vdash_{\hat{\Phi}} \langle \{i \hookrightarrow \hat{p}_i\}_{i \in L} \rangle \rightsquigarrow \mathtt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\Vdash \cup_{i \in L} \Gamma_i} \tag{B.8d}$$

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

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 $\cup_{i \in L} \hat{\Gamma}_i$ as shorthand for

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

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.6o) 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 stylized spTSM definition form is

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

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.6o) defines typed expansion of spTSM definitions:

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

This rule is similar to Rule (B.6l), 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.21.

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

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

   The type abbreviated `ProtoPat` 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 `ProtoPat` is defined by the *proto-pattern encoding judgement*, $\grave{p} \downarrow_{\text{ProtoPat}} e$. An inverse mapping is defined by the *proto-pattern decoding judgement*, $e \uparrow_{\text{ProtoPat}} \grave{p}$.

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

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

   Condition B.24 (Proto-Pattern Isomorphism)**.** *All of the following must hold:*

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

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

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

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

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

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

3. The final premise of Rule (B.6o) 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.6o) 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 $a \in \mathrm{dom}(\Phi)$ to an *expanded spTSM definition*, $a \hookrightarrow \mathtt{sptsm}(\tau; e_{\text{parse}})$, where τ is the spTSM's type annotation, and e_{parse} is its parse function. We write $\Phi, a \hookrightarrow \mathtt{sptsm}(\tau; e_{\text{parse}})$ when $a \notin \mathrm{dom}(\Phi)$ for the extension of Φ that maps a to $a \hookrightarrow \mathtt{sptsm}(\tau; e_{\text{parse}})$.

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

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

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:

$$\hat{a}\ /b/$$

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.

$$\frac{\begin{array}{c}\hat{\Phi} = \hat{\Phi}', \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{sptsm}(\tau; e_{\text{parse}}) \\ b \downarrow_{\mathsf{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \mathtt{SuccessP} \cdot e_{\text{proto}} \qquad e_{\text{proto}} \uparrow_{\mathsf{ProtoPat}} \grave{p} \\ \mathsf{seg}(\grave{p})\ \mathsf{segments}\ b \qquad \vdash^{\Delta; \hat{\Phi}; b} \grave{p} \rightsquigarrow p : \tau \dashv\!\!| \hat{\Gamma}\end{array}}{\Delta \vdash_{\hat{\Phi}} \hat{a}\ /b/ \rightsquigarrow p : \tau \dashv\!\!| \hat{\Gamma}} \qquad \text{(B.8f)}$$

where:

$$\mathtt{SuccessP} \cdot e \stackrel{\text{def}}{=} \mathtt{inj}[L_{\text{SP}}; \mathtt{SuccessP}]\{\mathtt{ParseError} \hookrightarrow \langle\rangle, \mathtt{SuccessP} \hookrightarrow \mathtt{ProtoPat}\}(e)$$

This rule is similar to Rule (B.6m), 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, i.e. a value of the form abbreviated $\texttt{SuccessP} \cdot e_{\text{proto}}$ (as shown) results from evaluation, then e_{proto} will be a value of type `ProtoPat` (assuming a well-formed spTSM context, by application of the Preservation assumption, Assumption B.10.) 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.6m) *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 3.7)
PrExp	$\grave{e}$	::=	$\cdots$	$\cdots$	(see Figure 3.7)
			$\texttt{prmatch}[n]\{\grave{\tau}\}(\grave{e}; \{\grave{r}_i\}_{1 \leq i \leq n})$	$\texttt{match}\ \grave{e}\ \{\grave{r}_i\}_{1 \leq i \leq n}$	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
			$\texttt{splicedp}[m; n]$	$\texttt{splicedp}\langle m, n \rangle$	spliced

Figure 4.4: Syntax of proto-expansion terms in miniVerse$_{\mathsf{S}}$.

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**
$\Delta \vdash^{\mathbb{T}} \grave{\tau} \rightsquigarrow \tau$ type	$\grave{\tau}$ has well-formed expansion τ
$\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau$	$\grave{e}$ has expansion e of type τ
$\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r} \rightsquigarrow r : \tau \Mapsto \tau'$	$\grave{r}$ has expansion r taking values of type τ to values of type τ'
$\vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : \tau \dashv\!\vert\vert^{\hat{\Gamma}}$	$\grave{p}$ 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 $\Delta; \hat{\Phi}; b$. 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.6m) 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*, $\Delta \vdash^{\mathbb{T}} \grave{\tau} \rightsquigarrow \tau$ 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*, $\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau$, and the *proto-rule validation judgement*, $\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r} \rightsquigarrow r : \tau \Mapsto \tau'$, 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.10l) were described in Sec. 3.2.10. There is one new common proto-expression form, for match proto-expressions, which is governed by Rule (B.10m):

$$\frac{\begin{array}{c}\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau \qquad \Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau}' \rightsquigarrow \tau' \text{ type} \\ \{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r}_i \rightsquigarrow r_i : \tau \Mapsto \tau'\}_{1 \le i \le n}\end{array}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prmatch}[n]\{\grave{\tau}'\}(\grave{e}; \{\grave{r}_i\}_{1 \le i \le n}) \rightsquigarrow \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \le i \le n}) : \tau'} \quad \text{(B.10m)}$$

Rule (B.11) governs proto-rules:

$$\frac{\Delta \vdash p : \tau \dashv\!\vert\vert \Gamma \qquad \Delta\ \Gamma \cup \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prrule}(p.\grave{e}) \rightsquigarrow \mathtt{rule}(p.e) : \tau \Mapsto \tau'} \quad \text{(B.11)}$$

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. It would not be sensible for an seTSM to splice patterns out of a literal body.

Proto-Pattern Validation

spTSMs generate candidate expansions of proto-pattern form, as described in Sec. 4.2.7. The *proto-pattern validation judgement*, $\vdash^{\mathbb{P}} \hat{p} \rightsquigarrow p : \tau \dashv\!|^{\hat{\Gamma}}$, which appears as the final premise of Rule (B.8f), validates proto-patterns by checking that the pattern matches values of type τ, and simultaneously generates the final expansion, p, and the hypotheses $\hat{\Gamma}$. Hypotheses can be generated only by spliced subpatterns, so there is no proto-pattern form corresponding to variable patterns (this is also why $\hat{\Gamma}$ appears as a superscript.)

The proto-pattern validation judgement is defined mutually inductively with Rules (B.8) by Rules (B.12), reproduced below.

$$\frac{}{\vdash^{\mathbb{P}} \texttt{prwildp} \rightsquigarrow \texttt{wildp} : \tau \dashv\!|^{\langle\emptyset;\emptyset\rangle}} \tag{B.12a}$$

$$\frac{\vdash^{\mathbb{P}} \hat{p} \rightsquigarrow p : [\texttt{rec}(t.\tau)/t]\tau \dashv\!|^{\hat{\Gamma}}}{\vdash^{\mathbb{P}} \texttt{prfoldp}(\hat{p}) \rightsquigarrow \texttt{foldp}(p) : \texttt{rec}(t.\tau) \dashv\!|^{\hat{\Gamma}}} \tag{B.12b}$$

$$\frac{\{\vdash^{\mathbb{P}} \hat{p}_i \rightsquigarrow p_i : \tau_i \dashv\!|^{\hat{\Gamma}_i}\}_{i\in L}}{\left(\begin{array}{c}\vdash^{\mathbb{P}} \texttt{prtplp}[L](\{i \hookrightarrow \hat{p}_i\}_{i\in L}) \\ \rightsquigarrow \\ \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i\in L}) : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L}) \dashv\!|^{\cup_{i\in L}\hat{\Gamma}_i}\end{array}\right)} \tag{B.12c}$$

$$\frac{\vdash^{\mathbb{P}} \hat{p} \rightsquigarrow p : \tau \dashv\!|^{\hat{\Gamma}}}{\vdash^{\mathbb{P}} \texttt{prinjp}[\ell](\hat{p}) \rightsquigarrow \texttt{injp}[\ell](p) : \texttt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i\in L}; \ell \hookrightarrow \tau) \dashv\!|^{\hat{\Gamma}}} \tag{B.12d}$$

$$\frac{\mathsf{parseUPat}(\mathsf{subseq}(b;m;n)) = \hat{p} \qquad \Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!|\, \hat{\Gamma}}{\vdash^{\Delta;\hat{\Phi};b} \texttt{splicedp}[m;n] \rightsquigarrow p : \tau \dashv\!|^{\hat{\Gamma}}} \tag{B.12e}$$

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 parses the indicated subsequence of the literal body, b, to produce the referenced unexpanded pattern, $\hat{p}$, and the second premise types and expands $\hat{p}$ under the spTSM context Φ from the spTSM application site, generating the hypotheses $\hat{\Gamma}$. These are the hypotheses generated in the conclusion of the rule.

Notice that none of these rules explicitly add any hypotheses to the pattern typing context, so spTSMs cannot introduce any hypotheses other than those that come from spliced patterns. This achieves the binding discipline described in Sec. 4.1.4.

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 same hypotheses. It must mutually state the corresponding proposition about proto-patterns, because the relevant judgements are mutually defined.

Theorem B.28 (Typed Pattern Expansion)**.** *Both of the following hold:*

1. *If* $\Delta \vdash_{\langle \mathcal{A};\Phi\rangle} \hat{p} \rightsquigarrow p : \tau \dashv\!| \langle \mathcal{G};\Gamma\rangle$ *then* $\Delta \vdash p : \tau \dashv\!| \Gamma$.
2. *If* $\vdash^{\Delta;\langle \mathcal{A};\Phi\rangle;b} \grave{p} \rightsquigarrow p : \tau \dashv\!|^{\langle \mathcal{G};\Gamma\rangle}$ *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 $\hat{\Gamma} = \langle \mathcal{G};\Gamma\rangle$ 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)	$e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{SuccessP} \cdot e_{\text{proto}}$	by assumption
(6)	$e_{\text{proto}} \uparrow_{\mathsf{ProtoPat}} \grave{p}$	by assumption
(7)	$\vdash^{\Delta;\langle \mathcal{A};\Phi\rangle;b} \grave{p} \rightsquigarrow p : \tau \dashv\!|^{\langle \mathcal{G};\Gamma\rangle}$	by assumption
(8)	$\Delta \vdash p : \tau \dashv\!| \Gamma$	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 $\hat{\Phi} = \langle \mathcal{A};\Phi\rangle$.

   Case (B.12e)**.**

(1)	$\grave{p} = \texttt{splicedp}[m;n]$	by assumption
(2)	$\mathsf{parseUExp}(\mathsf{subseq}(b;m;n)) = \hat{p}$	by assumption
(3)	$\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!| \hat{\Gamma}$	by assumption
(4)	$\Delta \vdash p : \tau \dashv\!| \Gamma$	by IH, part 1 on (3)

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:

$$\|\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!| \hat{\Gamma}\| = \|\hat{p}\|$$
$$\|\vdash^{\Delta;\hat{\Phi};b} \grave{p} \rightsquigarrow p : \tau \dashv\!|^{\hat{\Gamma}}\| = \|b\|$$

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:

$$\begin{aligned}&\|\Delta \vdash_{\hat{\Phi}} \hat{a}\ /b/ \rightsquigarrow p : \tau \dashv\!| \hat{\Gamma}\| \\ =&\|\vdash^{\Delta;\hat{\Phi};b} \grave{p} \rightsquigarrow p : \tau \dashv\!|^{\hat{\Gamma}}\| \\ =&\|b\|\end{aligned}$$

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 $\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!| \hat{\Gamma}$. 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:

$$\|\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!| \hat{\Gamma}\| < \|\vdash^{\Delta;\hat{\Phi};b} \mathtt{splicedp}[m; n] \rightsquigarrow p : \tau \dashv\!|^{\hat{\Gamma}}\|$$

i.e. by the definitions above,

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

This is established by appeal to Condition B.22, 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.14 (Pattern Parsing Monotonicity)**.** *If* $\mathsf{parseUPat}(b) = \hat{p}$ *then* $\|\hat{p}\| < \|b\|$.

Combining Conditions B.22 and B.14, 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)**.** *All of the following hold:*

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.4.3. 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.6n)**.**

(1)	$\hat{e} = \mathtt{match}\ \hat{e}'\ \{\hat{r}_i\}_{1 \le i \le n}$	by assumption
(2)	$e = \mathtt{match}[n]\{\tau\}(e'; \{r_i\}_{1 \le i \le n})$	by assumption
(3)	$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \rightsquigarrow e' : \tau'$	by assumption
(4)	$\Delta \vdash \tau\ \mathsf{type}$	by assumption
(5)	$\{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \rightsquigarrow r_i : \tau' \Mapsto \tau\}_{1 \le i \le n}$	by assumption
(6)	$\Delta\ \Gamma \vdash e' : \tau'$	by IH, part 1(a) on (3)

(7) $\{\Delta\ \Gamma \vdash r_i : \tau' \Rightarrow\!\!\!\Rightarrow \tau\}_{1 \leq i \leq n}$ by IH, part 1(b) over (5)

(8) $\Delta\ \Gamma \vdash \texttt{match}[n]\{\tau\}(e'; \{r_i\}_{1 \leq i \leq n}) : \tau$ by Rule (B.2l) on (6), (4) and (7)

Case (B.6o).

(1) $\hat{e} = \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}'\ \texttt{for patterns by static}\ e_{\text{parse}}\ \texttt{end in}\ \hat{e}'$ by assumption

(2) $\hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'\ \text{type}$ by assumption

(3) $\emptyset\ \emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResultSP)}$ by assumption

(4) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi},\hat{a} \rightsquigarrow a \hookrightarrow \texttt{sptsm}(\tau'; e_{\text{parse}})} \hat{e}' \rightsquigarrow e : \tau$ by assumption

(5) $\Delta \vdash \tau'\ \text{type}$ by Lemma B.26 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) $\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \langle \mathcal{A}'; \Gamma' \rangle$ by assumption

(4) $\hat{\Delta}\ \langle \mathcal{A} \uplus \mathcal{A}'; \Gamma \cup \Gamma' \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau'$ by assumption

(5) $\Delta \vdash p : \tau \dashv\!\mid \Gamma'$ by Theorem B.28, part 1 on (3)

(6) $\Delta\ \Gamma \cup \Gamma' \vdash e : \tau'$ by IH, part 1(a) on (4)

(7) $\Delta\ \Gamma \vdash \texttt{rule}(p.e) : \tau \Rightarrow\!\!\!\Rightarrow \tau'$ 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.10m).

(1) $\grave{e} = \texttt{prmatch}[n]\{\grave{\tau}\}(\grave{e}'; \{\grave{r}_i\}_{1 \leq i \leq n})$ by assumption

(2) $e = \texttt{match}[n]\{\tau\}(e'; \{r_i\}_{1 \leq i \leq n})$ by assumption

(3) $\Delta\ \Gamma \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e}' \rightsquigarrow e' : \tau'$ by assumption

(4) $\Delta \vdash^{\hat{\Delta};b} \grave{\tau} \rightsquigarrow \tau\ \text{type}$ by assumption

(5) $\{\Delta\ \Gamma \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{r}_i \rightsquigarrow r_i : \tau' \Rightarrow\!\!\!\Rightarrow \tau\}_{1 \leq i \leq n}$ 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) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash e' : \tau'$ by IH, part 2(a) on (3), (6) and (7)

(9) $\Delta \cup \Delta_{\text{app}} \vdash \tau\ \text{type}$ by Lemma B.27 on (4)

(10) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash r : \tau' \Rightarrow\!\!\!\Rightarrow \tau$ by IH, part 2(b) on (5), (6) and (7)

(11) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash \texttt{match}[n]\{\tau\}(e'; \{r_i\}_{1 \leq i \leq n}) : \tau$

by Rule (B.2l) on (8), (9), (10)

(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) $\Delta \vdash p : \tau \dashv\Vert\, \Gamma$ — 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) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash \texttt{rule}(p.e) : \tau \Rightarrow\!\!\!\!\mid\ \tau'$ — 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.4.3 gives the complete details. □

Reasoning Principles

The following theorem, together with Theorem B.28 part 1, establishes **Typing** and **Segmentation**, as discussed in Sec. 4.1.4.

Theorem B.33 (spTSM Typing and Segmentation). *If* $\Delta \vdash_{\hat{\Phi}} \hat{a}\ /b/ \rightsquigarrow p : \tau \dashv\Vert\, \hat{\Gamma}$ *then*

1. *(**Typing**)* $\hat{\Phi} = \hat{\Phi}', \hat{a} \rightsquigarrow a \hookrightarrow \texttt{sptsm}(\tau; e_{parse})$
2. $b \downarrow_{\text{Body}} e_{body}$
3. $e_{parse}(e_{body}) \Downarrow \texttt{SuccessP} \cdot e_{proto}$
4. $e_{proto} \uparrow_{\text{ProtoPat}} \grave{p}$
5. *(**Segmentation**)* $\text{seg}(\grave{p})$ segments b

Proof. By rule induction over Rules (B.8). The only rule that applies is Rule (B.8f). The conclusions are premises of this rule. □

Part II

Parametric TSMs

You know me, I gotta put in a big tree.

— Bob Ross, *The Joy of Painting*

Chapter 5

Parametric TSMs (pTSMs)

In the two previous chapters, we introduced simple TSMs (sTSMs). Simple TSMs are defined at a single type, like `rx`, and the expansions that they generate have access to bindings at the application site only through spliced terms that the client provides. In this chapter, we introduce *parametric TSMs* (pTSMs). pTSMs can be defined over a type- and module-parameterized family of types, and the expansions that they generate can refer to the supplied type and module parameters.

This chapter is organized like the preceding chapters. We begin in Sec. 5.1 by introducing parameterized 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, 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)
```

ML dialects commonly define derived syntactic forms for constructing and pattern matching over values of list type. VerseML, in contrast, does not build in derived list forms. Instead, the provider of the library where the type function `list` is defined also defines a *parametric expression TSM* (peTSM) and a *parametric pattern TSM* (ppTSM), both named `$list`, as shown in Figure 5.1.

```
syntax $list('a) at list('a) for expressions by
  static fn(b : body) : parse_result(proto_expr) => (* ... *)
and for patterns by
  static fn(b : body) : parse_result(proto_pat) => (* ... *)
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 `$list`, when applied to a type `T` and a generalized literal form, can only generate expansions (of type `/` that match values of type) `list(T)`. For example, we can apply `$list` to `int` and generalized literal forms delimited by square brackets as follows:

```
val y = $list(int) [3, 4, 5]
val x = $list(int) [1, 2 :: y]
```

Parse functions operate as described in Chapter 3 to generate encodings of proto-expansions, which are subsequently validated to generate the final expansions of expressions of TSM application form. For `$list`, the parse functions (whose definitions are elided above for concision) break the literal body up into spliced terms – those separated by commas become individual elements at the head of the list, and, optionally, a trailing spliced term prefixed by two colons (`::`) becomes 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 y = Cons(3, Cons(4, Cons(5, Nil)))
val x = Cons(1, Cons(2, y))
```

Once again, due to the prohibition on context-dependent expansions, the expansion itself must use the explicit **`fold`** and **`inj`** operators rather than the list value constructors `Cons` and `Nil`.

5.1.2 Module Parameters

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.) Let us now define parameterized TSM for lists, shown in Figure 5.2, that take modules matching `LIST` as an additional parameter.

```
syntax $list' (L : LIST) 'a at 'a L.list for expressions by
  static fn(b : body) : parse_result(proto_expr) => (* ... *)
and 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 constuctors are projected out of the provided module parameter. The TSM itself did not assume that the module would be named `List` (indeed, internally, it 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`.

```
1 syntax $r(R : RX) at R.t by
2   static fn(b : body) : parse_result(proto_expr) => (* ... *)
3 end
```

Figure 5.3: The module-parameterized TSM `$r`.

To be clear: TSM parameters are available only to (proto-)expansions. Parameters are not available to the parse function. 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
```

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. This language, like miniVerse$_{\mathbf{S}}$, consists of an unexpanded language (UL) defined by typed expansion to an expanded language (XL).

5.2.1 Syntax of the Expanded Language (XL)

The miniVerse$_{\mathbf{P}}$ XL extends the miniVerse$_{\mathbf{S}}$ XL with a *module language* and a *type construction language*. Both of these are based directly on the languages defined in *PFPL* [47]. This language, in turn, is similar to the internal language defined by Lee et al. [62], and also to the internal language used by Dreyer [27]. Both of these incorporate Stone and Harper's *dependent singleton kinds* formalism to track type identity [93].

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	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	::=	$\texttt{darr}(\kappa;u.\kappa)$	dependent function
			$\texttt{unit}$	nullary product
			$\texttt{dprod}(\kappa;u.\kappa)$	dependent product
			$\texttt{Type}$	types
			$\texttt{S}(\tau)$	singleton
Con	c,τ	::=	u	variable
			t	
			$\texttt{abs}(u.c)$	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
			$\texttt{con}(M)$	constructor part

Figure 5.5: Syntax of kinds and constructors in miniVerse$_\mathbf{P}$. By convention, we choose the metavariable τ for constructors that, in well-formed terms, must necessarily be of kind $\texttt{T}$, and the metavariable c otherwise. Similarly, we use constructor variables t to stand for constructors of kind $\texttt{T}$, and constructor variables u otherwise.

Sort			Operational Form	Description
Exp	e	::=	x	variable
			$\texttt{lam}\{\tau\}(x.e)$	abstraction
			$\texttt{ap}(e;e)$	application
			$\texttt{clam}\{\kappa\}(u.e)$	constructor abstraction
			$\texttt{cap}\{\kappa\}(e)$	constructor application
			$\texttt{fold}\{t.\tau\}(e)$	fold
			$\texttt{unfold}(e)$	unfold
			$\texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L})$	labeled tuple
			$\texttt{prj}[\ell](e)$	projection
			$\texttt{inj}[L;\ell]\{\{i \hookrightarrow \tau_i\}_{i\in L}\}(e)$	injection
			$\texttt{match}[n]\{\tau\}(e;\{r_i\}_{1\le i\le n})$	match
			$\texttt{val}(M)$	value part
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 *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
$\Omega \vdash \sigma$ sig	σ is a signature
$\Omega \vdash \sigma \equiv \sigma'$	σ and σ' are definitionally equal
$\Omega \vdash \sigma <: \sigma'$	σ is a sub-signature of σ'
$\Omega \vdash M : \sigma$	M matches σ
$\Omega \vdash M$ mval	M is, or stands for, a module value

The second group of judgements, which we refer to as the *statics of the expanded constructor language*, define the statics of expanded kinds and constructors.

Judgement Form	Description
$\Omega \vdash \kappa$ kind	κ is a kind
$\Omega \vdash \kappa \equiv \kappa'$	κ and κ' are equivalent
$\Omega \vdash \kappa <:: \kappa'$	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 constructors 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.

Judgement Form	**Description**
$\Omega \vdash \tau$ type	τ is a well-formed type
$\Omega \vdash \tau \equiv \tau'$ type	τ and τ' are equivalent types
$\Omega \vdash \tau <: \tau'$	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'
$\Omega \vdash p : \tau \dashv\Vdash \Omega'$	p matches values of type τ and generates hypotheses Ω'

A *unified context*, Ω, is a finite function. We write

- $\Omega, x : \tau$ when $x \notin \mathrm{dom}(\Omega)$ and $\Omega \vdash \tau :: \mathtt{Type}$ for the extension of Ω with a mapping from x to the hypothesis $x : \tau$
- $\Omega, u :: \kappa$ when $u \notin \mathrm{dom}(\Omega)$ and $\Omega \vdash \kappa$ kind for the extension of Ω with a mapping from u to the hypothesis $u :: \kappa$
- $\Omega, X : \sigma$ when $X \notin \mathrm{dom}(\Omega)$ and $\Omega \vdash \sigma$ sig for the extension of Ω with a mapping from X to the hypothesis $X : \sigma$.

A well-formed unified context is one that can be constructed by some sequence of such extensions, start 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 [47].) Instead, let us summarize the key features of the expanded language by example.

We encode modules using a *phase-splitting* approach where the constructor components are "tupled" into a single constructor component and the value components are "tupled" into a single value component [48]. Signatures, σ, classify module expressions, and are also split in this way – a single *kind* classifies the constructor component and a single type classifies the value component of the classified module. The type in the signature can refer to the constructor component of the module through a mediating constructor variable. The key rule is reproduced below:

$$\frac{\Omega \vdash c :: \kappa \qquad \Omega \vdash e : [c/u]\tau}{\Omega \vdash \mathtt{struct}(c; e) : \mathtt{sig}\{\kappa\}(u.\tau)} \tag{C.4c}$$

For example, consider the following VerseML signature on the left, and the corresponding miniVerse$_\mathbf{P}$ signature on the right:

```
1 sig
2   type t
3   type t' = t * t
4   val x : t
5   val y : t -> t'
6 end
```

```
sig{dprod(
    Type; t.
    S(prod[1;2](1 ↪ t; 2 ↪ t)))
}(u.prod[x;y](x ↪ prl(u);
    y ↪ parr(prl(u); prr(u)))
)
```

The kind in the signature (Lines 1-3, right) is a *dependent product kind* and the type (Lines 4-5, right) is a product type. Let us consider these in turn.

On Lines 2-3 (left), we specified an abstract type component `t`, and then a translucent type component `t'`. Abstract type components have kind `Type`, as seen on Line 2 (right). The constructor variable t stands for this abstract type component in the right side 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`, as shown on Line 3 (right). A singleton kind $\mathtt{S}(\tau)$ classifies only those types definitionally equal to τ (whereas the kind `Type` classifies all types.) A subkinding relation is necessary to ensure that constructors of singleton kind can appear where a constructor of kind `Type` is needed – the key rule is reproduced below:

$$\frac{\Omega \vdash \tau :: \mathtt{Type}}{\Omega \vdash \mathtt{S}(\tau) <:: \mathtt{Type}} \tag{C.8e}$$

Moving on, Lines 4-5 (right) define a product type that classifies the value component of matching modules. The constructor variable `u` stands for the constructor component of the matching module. Because the constructor component is of dependent product kind, we must use the left- and right-projection operators, $\mathtt{prl}(c)$ and $\mathtt{prr}(c)$. (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:

```
1 sig
2   type list('a) = rec(self => Nil + Cons of 'a * self)
3   val Nil : list('a)
4   val Cons : 'a * list('a) -> list('a)
5   (* ... *)
6 end
```

This VerseML signature corresponds to the following miniVerse$_\mathbf{P}$ signature:

$$\begin{aligned}
\sigma_{\mathtt{LIST}} &\stackrel{\text{def}}{=} \mathtt{sig}\{\kappa_{\mathtt{LIST}}\}(\mathit{list}.\tau_{\mathtt{LIST}}) \\
\kappa_{\mathtt{LIST}} &\stackrel{\text{def}}{=} \mathtt{darr}(\mathtt{Type}; \alpha.\mathtt{S}(\mathtt{rec}(\mathit{self}.\mathtt{sum}[L_{\mathtt{list}}](\\
&\qquad \mathtt{Nil} \hookrightarrow \mathtt{prod}[](); \\
&\qquad \mathtt{Cons} \hookrightarrow \mathtt{prod}[1;2](1 \hookrightarrow \alpha; 2 \hookrightarrow \mathit{self})))))\\
L_{\mathtt{list}} &\stackrel{\text{def}}{=} \mathtt{Nil}, \mathtt{Cons} \\
\tau_{\mathtt{LIST}} &\stackrel{\text{def}}{=} \mathtt{prod}[L_{\mathtt{list}}](\\
&\qquad \mathtt{Nil} \hookrightarrow \mathtt{all}\{\mathtt{Type}\}(\alpha.\mathtt{app}(\mathit{list}; \alpha)); \\
&\qquad \mathtt{Cons} \hookrightarrow \mathtt{all}\{\mathtt{Type}\}(\alpha.\mathtt{parr}(\\
&\qquad\quad \mathtt{prod}[1;2](1 \hookrightarrow \alpha; 2 \hookrightarrow \mathtt{app}(\mathit{list}; \alpha)); \\
&\qquad\quad \mathtt{app}(\mathit{list}; \alpha))))
\end{aligned}$$

Here, there is only a single constructor component, so a dependent product kind is not needed. This component is a type function, so it has dependent function kind. The argument is of kind `Type` and the return kind is a singleton kind, because the type function is not abstract. Had we held the list type function abstract, its kind would instead be:

$$\mathtt{darr}(\mathtt{Type}; \_.\mathtt{Type})$$

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:

$$\frac{\Omega \vdash M : \sigma \qquad \Omega \vdash \sigma' \ \mathsf{sig} \qquad \Omega, X : \sigma \vdash M' : \sigma'}{\Omega \vdash \mathtt{mlet}\{\sigma'\}(M; X.M') : \sigma'} \tag{C.4e}$$

To be able to use the constructor defined by a module, M, within a constructor appearing in some other module, we need a constructor projection form, $\mathtt{con}(M)$. The kinding rule for this form is reproduced below:

$$\frac{\Omega \vdash M \ \mathsf{mval} \qquad \Omega \vdash M : \mathtt{sig}\{\kappa\}(u.\tau)}{\Omega \vdash \mathtt{con}(M) :: \kappa} \tag{C.9o}$$

Similarly, the value projection form, $\mathtt{val}(M)$, projects out the value component of M within an expression. The typing rule for this form is reproduced below:

$$\frac{\Omega \vdash M \ \mathsf{mval} \qquad \Omega \vdash M : \mathtt{sig}\{\kappa\}(u.\tau)}{\Omega \vdash \mathtt{val}(M) : [\mathtt{con}(M)/u]\tau} \tag{C.14m}$$

The first premise of both of these rules requires that M be, or stand for, a *module value*, according to the following rules:

$$\frac{}{\Omega \vdash \mathtt{struct}(c; e) \ \mathsf{mval}} \tag{C.5a}$$

$$\frac{}{\Omega, X : \sigma \vdash X \ \mathsf{mval}} \tag{C.5b}$$

The reason for this restriction has to do with the *sealing* operation:

$$\frac{\Omega \vdash \sigma \ \mathsf{sig} \qquad \Omega \vdash M : \sigma}{\Omega \vdash \mathtt{seal}\{\sigma\}(M) : \sigma} \tag{C.4d}$$

Sealing is intended to support *representation independence*, i.e. the abstract constructor components of a sealed module are not equivalent to those of any other sealed module. In other words, sealing is *generative*. The module value restriction above encodes this behavior by simple syntactic means – a sealed module is not a module value, so it has to first be bound to a module variable for it to be possible to refer to its constructor components. Different bindings, even if of the same underlying module, induce distinct constructor projections.

The judgements above obey standard lemmas, including Weakening, Substitution and Decomposition (see Appendix C.1.2.)

Some aspects of the ML module system are not available in miniVerse$_\mathbf{P}$, including its support for hierarchical modules and functors. Also, our formulation 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 for concision.

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

Definition C.5 (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, 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.6 (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.7 (Progress)**.**

1. *If $\vdash M : \sigma$ then either M* val *or M* matchfail *or there exists an M' such that $M \mapsto M'$.*
2. *If $\vdash e : \tau$ then either e* val *or e* matchfail *or there exists an e' such that $e \mapsto e'$.*

5.2.4 Syntax of the Surface Language

Syntax of the Surface Core Language

Unexpanded kinds and constructors in Figure 5.7

Unexpanded expressions, rules and patterns in Figure 5.8

Sort			Operational Form	Stylized Form	Description
UKind	$\hat{\kappa}$	$::=$	$\mathtt{udarr}(\hat{\kappa}; \hat{u}.\hat{\kappa})$	$(\hat{u} :: \hat{\kappa}) \to \hat{\kappa}$	dependent function
			$\mathtt{uunit}$	$\langle\!\rangle$	nullary product
			$\mathtt{udprod}(\hat{\kappa}; \hat{u}.\hat{\kappa})$	$(\hat{u} :: \hat{\kappa}) \times \hat{\kappa}$	dependent product
			$\mathtt{uType}$	$\mathtt{T}$	types
			$\mathtt{uS}(\hat{\tau})$	$[{=}\hat{\tau}]$	singleton
UCon	$\hat{c}, \hat{\tau}$	$::=$	$\hat{u}$	$\hat{u}$	constructor sigil
			$\hat{t}$	$\hat{t}$	constructor sigil
			$\mathtt{uasc}\{\hat{\kappa}\}(\hat{c})$	$\hat{c} :: \hat{\kappa}$	ascription
			$\mathtt{uabs}(\hat{u}.\hat{c})$	$\lambda \hat{u}.\hat{c}$	abstraction
			$\mathtt{uapp}(c; c)$	$c(c)$	application
			$\mathtt{utriv}$	$\langle\!\rangle$	trivial
			$\mathtt{upair}(\hat{c}; \hat{c})$	$\langle\!\langle \hat{c}, \hat{c} \rangle\!\rangle$	pair
			$\mathtt{uprl}(\hat{c})$	$\hat{c} \cdot \mathtt{l}$	left projection
			$\mathtt{uprr}(\hat{c})$	$\hat{c} \cdot \mathtt{r}$	right projection
			$\mathtt{uparr}(\hat{\tau}; \hat{\tau})$	$\hat{\tau} \rightharpoonup \hat{\tau}$	partial function
			$\mathtt{uall}\{\hat{\kappa}\}(\hat{u}.\hat{\tau})$	$\forall(\hat{u} :: \hat{\kappa}).\hat{\tau}$	polymorphic
			$\mathtt{urec}(\hat{t}.\hat{\tau})$	$\mu \hat{t}.\hat{\tau}$	recursive
			$\mathtt{uprod}[L](\{i \hookrightarrow \hat{\tau}_i\}_{i \in L})$	$\langle \{i \hookrightarrow \hat{\tau}_i\}_{i \in L} \rangle$	labeled product
			$\mathtt{usum}[L](\{i \hookrightarrow \hat{\tau}_i\}_{i \in L})$	$[\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}]$	labeled sum
			$\mathtt{ucon}(\hat{X})$	$\hat{X} \cdot \mathtt{c}$	constructor part

Figure 5.7: Syntax of unexpanded kinds and constructors in miniVerse$_{\mathbf{P}}$. By convention, we choose the metavariable $\hat{\tau}$ for constructors that, in well-formed terms, must necessarily expand to constructors of kind $\mathtt{T}$, and the metavariable $\hat{c}$ otherwise. Similarly, we choose metavariables $\hat{t}$ for constructor sigils that expand to constructor variables that stand for constructors of kind $\mathtt{T}$, and constructor sigils $\hat{u}$ otherwise. Unexpanded kinds and constructors are not identified up to α-equivalence.

Sort			Operational Form	Stylized Form	Description
UExp	$\hat{e}$	::=	$\hat{x}$	$\hat{x}$	sigil
			$\texttt{uasc}\{\hat{\tau}\}(\hat{e})$	$\hat{e} : \hat{\tau}$	ascription
			$\texttt{uletval}(\hat{e}; \hat{x}.\hat{e})$	$\texttt{let val}\ \hat{x} = \hat{e}\ \texttt{in}\ \hat{e}$	value binding
			$\texttt{uanalam}(\hat{x}.\hat{e})$	$\lambda\hat{x}.\hat{e}$	abstraction (unannotated)
			$\texttt{ulam}\{\hat{\tau}\}(\hat{x}.\hat{e})$	$\lambda\hat{x}{:}\hat{\tau}.\hat{e}$	abstraction (annotated)
			$\texttt{uap}(\hat{e}; \hat{e})$	$\hat{e}(\hat{e})$	application
			$\texttt{uclam}\{\hat{\kappa}\}(\hat{u}.\hat{e})$	$\Lambda\hat{u}{::}\hat{\kappa}.\hat{e}$	constructor abstraction
			$\texttt{ucap}\{\hat{c}\}(\hat{e})$	$\hat{e}[\hat{c}]$	constructor application
			$\texttt{ufold}(\hat{e})$	$\texttt{fold}(\hat{e})$	fold
			$\texttt{uunfold}(\hat{e})$	$\texttt{unfold}(\hat{e})$	unfold
			$\texttt{utpl}\{L\}(\{i \hookrightarrow \hat{e}_i\}_{i \in L})$	$\langle \{i \hookrightarrow \hat{e}_i\}_{i \in L} \rangle$	labeled tuple
			$\texttt{uprj}[\ell](\hat{e})$	$\hat{e} \cdot \ell$	projection
			$\texttt{uin}[\ell](\hat{e})$	$\texttt{inj}[\ell](\hat{e})$	injection
			$\texttt{umatch}[n](\hat{e}; \{\hat{r}_i\}_{1 \leq i \leq n})$	$\texttt{match}\ \hat{e}\ \{\hat{r}_i\}_{1 \leq i \leq n}$	match
			$\texttt{uval}(\hat{X})$	$\hat{X} \cdot \texttt{v}$	value part
			$\texttt{usyntaxpe}\{e\}\{\hat{\rho}\}(\hat{a}.\hat{e})$	$\texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\rho}\ \texttt{for expressions}\ \{e\}\ \texttt{in}\ \hat{e}$	peTSM definition
			$\texttt{uletpetsm}\{\hat{e}\}(\hat{a}.\hat{e})$	$\texttt{let syntax}\ \hat{a} = \hat{e}\ \texttt{for expressions in}\ \hat{e}$	peTSM binding
			...	...	peTSM designation
			$\texttt{uappetsm}[b]\{\hat{e}\}$	$\hat{e}\ /b/$	peTSM application
			$\texttt{uelit}[b]$	$/b/$	peTSM unadorned literal
			$\texttt{usyntaxpp}\{e\}\{\hat{\rho}\}(\hat{a}.\hat{e})$	$\texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\rho}\ \texttt{for patterns}\ \{e\}\ \texttt{in}\ \hat{e}$	ppTSM definition
			$\texttt{uletpptsm}\{\hat{e}\}(\hat{a}.\hat{e})$	$\texttt{let syntax}\ \hat{a} = \hat{e}\ \texttt{for patterns in}\ \hat{e}$	ppTSM binding
			...	...	ppTSM designation
URule	$\hat{r}$	::=	$\texttt{urule}(\hat{p}.\hat{e})$	$\hat{p} \Rightarrow \hat{e}$	match rule
UPat	$\hat{p}$	::=	$\hat{x}$	$\hat{x}$	sigil pattern
			$\texttt{uwildp}$	$\_$	wildcard pattern
			$\texttt{ufoldp}(\hat{p})$	$\texttt{fold}(\hat{p})$	fold pattern
			$\texttt{utplp}[L](\{i \hookrightarrow \hat{p}_i\}_{i \in L})$	$\langle \{i \hookrightarrow \hat{p}_i\}_{i \in L} \rangle$	labeled tuple pattern
			$\texttt{uinjp}[\ell](\hat{p})$	$\texttt{inj}[\ell](\hat{p})$	injection pattern
			$\texttt{uappptsm}[b]\{\hat{e}\}$	$\hat{e}\ /b/$	ppTSM application
			$\texttt{uplit}[b]$	$/b/$	ppTSM unadorned literal

Figure 5.8: Abstract syntax of unexpanded expressions, rules and patterns in miniVerse$_{\mathbf{P}}$.

Sort			Operational Form	Stylized Form	Description
USig	$\hat{\sigma}$	::=	`usig{`$\hat{\kappa}$`}(`$\hat{u}.\hat{\tau}$`)`	$[\![\hat{u} :: \hat{\kappa}; \hat{\tau}]\!]$	signature
UMod	$\hat{M}$	::=	$\hat{X}$	$\hat{X}$	module sigil
			`ustruct(`$\hat{c}; \hat{e}$`)`	$[\![\hat{c}; \hat{e}]\!]$	structure
			`useal{`$\hat{\sigma}$`}(`$\hat{M}$`)`	$\hat{M} \upharpoonright \hat{\sigma}$	seal
			`umlet{`$\hat{\sigma}$`}(`$\hat{M}; \hat{X}.\hat{M}$`)`	$(\texttt{let}\ \hat{X} = \hat{M}\ \texttt{in}\ \hat{M}) : \hat{\sigma}$	definition
			`umsyntaxpe{`e`}{`$\hat{\rho}$`}(`$\hat{a}.\hat{M}$`)`	`syntax` $\hat{a}$ `at` $\hat{\rho}$ `for expressions` $\{e\}$ `in` $\hat{M}$	peTSM definition
			`umletpetsm{`$\hat{e}$`}(`$\hat{a}.\hat{M}$`)`	`let syntax` $\hat{a} = \hat{e}$ `for expressions in` $\hat{M}$	peTSM binding
			...	...	peTSM designation
			`usyntaxpp{`e`}{`$\hat{\rho}$`}(`$\hat{a}.\hat{M}$`)`	`syntax` $\hat{a}$ `at` $\hat{\rho}$ `for patterns` $\{e\}$ `in` $\hat{M}$	ppTSM definition
			`uletpptsm{`$\hat{e}$`}(`$\hat{a}.\hat{M}$`)`	`let syntax` $\hat{a} = \hat{e}$ `for patterns in` $\hat{M}$	ppTSM binding
			...	...	ppTSM designation

Figure 5.9: Abstract syntax of unexpanded module expressions and signatures in miniVerse$_\mathbf{P}$.

Sort			Operational Form	Stylized Form	Description
MType	ρ	::=	`type(`τ`)`	τ	type annotation
			`alltypes(`$t.\rho$`)`	$\forall t.\rho$	type abstraction
			`allmods{`σ`}(`$X.\rho$`)`	$\forall X{:}\sigma.\rho$	module abstraction
UMType	$\hat{\rho}$	::=	`utype(`$\hat{\tau}$`)`	$\hat{\tau}$	type annotation
			`ualltypes(`$\hat{t}.\hat{\rho}$`)`	$\forall \hat{t}.\hat{\rho}$	type abstraction
			`uallmods{`$\hat{\sigma}$`}(`$\hat{X}.\hat{\rho}$`)`	$\forall \hat{X}{:}\hat{\sigma}.\hat{\rho}$	module abstraction
MExp	ϵ	::=	`defref[`a`]`	a	TSM definition reference
			`abstype(`$t.\epsilon$`)`	$\Lambda t.\epsilon$	type abstraction
			`absmod{`σ`}(`$X.\epsilon$`)`	$\Lambda X{:}\sigma.\epsilon$	module abstraction
			`aptype{`τ`}(`ϵ`)`	$\epsilon(\tau)$	type application
			`apmod{`M`}(`ϵ`)`	$\epsilon(M)$	module application
UMExp	$\hat{\epsilon}$	::=	`bindref[`$\hat{a}$`]`	$\hat{a}$	TSM binding reference
			`uabstype(`$\hat{t}.\hat{\epsilon}$`)`	$\Lambda \hat{t}.\hat{\epsilon}$	type abstraction
			`uabsmod{`$\hat{\sigma}$`}(`$\hat{X}.\hat{\epsilon}$`)`	$\Lambda \hat{X}{:}\hat{\sigma}.\hat{\epsilon}$	module abstraction
			`uaptype{`$\hat{\tau}$`}(`$\hat{\epsilon}$`)`	$\hat{\epsilon}(\hat{\tau})$	type application
			`uapmod{`$\hat{M}$`}(`$\hat{\epsilon}$`)`	$\hat{\epsilon}(\hat{M})$	module application

Figure 5.10: Abstract syntax of TSM types, unexpanded TSM types, TSM expressions and unexpanded TSM expressions in miniVerse$_\mathbf{P}$.

Syntax of the TSM Language

Macro types, unexpanded macro types, macro expressions and unexpanded macro expressions in Figure 5.10

Syntax of Type Patterns

TODO: do this

Syntax of the Surface Module Language

Figure 5.9

5.2.5 Typed Expansion

Unexpanded kinds and constructors

Judgement Form	Description
$\hat{\Omega} \vdash \hat{\kappa} \rightsquigarrow \kappa$ kind	$\hat{\kappa}$ is well-formed and has expansion κ
$\hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Rightarrow \kappa$	$\hat{c}$ has expansion c and synthesizes kind κ
$\hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Leftarrow \kappa$	$\hat{c}$ has expansion c when analyzed against kind κ

Unexpanded types, expressions, rules and patterns

Judgement Form	Description
$\hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau$ type	$\hat{\tau}$ has expansion τ
$\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau$	$\hat{e}$ has expansion e and synthesizes type τ
$\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau$	$\hat{e}$ has expansion e when analyzed against type τ
$\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \rightsquigarrow r \Rightarrow \tau \Mapsto \tau'$	$\hat{r}$ has expansion r and takes values of type τ to values of synthesized type τ'
$\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \rightsquigarrow r \Leftarrow \tau \Mapsto \tau'$	$\hat{r}$ has expansion r and takes values of type τ to values of type τ' when $\tau's$ is provided for analysis
$\hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\vert\vert \hat{\Omega}$	$\hat{p}$ has expansion p and type τ and generates hypotheses $\hat{\Omega}$

Unexpanded signatures and module expressions

Judgement Form	Description
$\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma$ sig	$\hat{\sigma}$ has expansion σ
$\hat{\Omega} \vdash_{\hat{\Psi}_E;\hat{\Psi}_P} \hat{M} \rightsquigarrow M \Rightarrow \sigma$	$\hat{M}$ has expansion M and synthesizes signature σ
$\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{M} \rightsquigarrow M \Leftarrow \sigma$	$\hat{M}$ has expansion M and matches signature σ

A *unified outer context*, $\hat{\Omega}$, takes the form $\langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega \rangle$, where $\mathcal{D}$ is a *constructor sigil expansion context*, $\mathcal{G}$ is an *expression sigil expansion context*, $\mathcal{M}$ is a *module sigil expansion context* and Ω is a unified inner context.

A constructor sigil expansion context, $\mathcal{D}$, is a finite function that maps each constructor sigil $\hat{u} \in \mathrm{dom}(\mathcal{D})$ to the constructor sigil expansion $\hat{u} \rightsquigarrow u$. We write $\hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa$ when $\hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega \rangle$ as an abbreviation of

$$\langle \mathcal{D} \uplus \hat{u} \rightsquigarrow u; \mathcal{G}; \mathcal{M}; \Omega, u :: \kappa \rangle$$

An expression sigil expansion context, $\mathcal{G}$, is a finite function that maps each expression sigil $\hat{x} \in \mathrm{dom}(\mathcal{G})$ to the expression sigil expansion $\hat{x} \rightsquigarrow x$. We write $\hat{\Omega}, \hat{x} \rightsquigarrow x : \tau$ when $\hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega \rangle$ as an abbreviation of

$$\langle \mathcal{D}; \mathcal{G} \uplus \hat{x} \rightsquigarrow x; \mathcal{M}; \Omega, x : \tau \rangle$$

A module sigil expansion context, $\mathcal{M}$, is a finite function that maps each module sigil $\hat{X} \in \mathrm{dom}(\mathcal{M})$ to the module sigil expansion $\hat{X} \rightsquigarrow X$. We write $\hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma$ when $\hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega \rangle$ as an abbreviation of

$$\langle \mathcal{D}; \mathcal{G}; \mathcal{M} \uplus \hat{X} \rightsquigarrow X; \Omega, X : \sigma \rangle$$

A *peTSM context*, $\hat{\Psi}$, takes the form $\langle \mathcal{A}; \Psi; \mathcal{I} \rangle$ where $\mathcal{A}$ is a *TSM binding context*, Ψ is a *peTSM definition context*, and $\mathcal{I}$ is TODO: implicits. Similarly, a *ppTSM context*, $\hat{\Phi}$, takes the form $\langle \mathcal{A}; \Phi; \mathcal{I} \rangle$ where Φ is a *ppTSM definition context*.

A *TSM binding context*, $\mathcal{A}$, is a finite function that maps each TSM name $\hat{a} \in \mathrm{dom}(\mathcal{A})$ to a *TSM binding*, $\hat{a} \hookrightarrow \epsilon$, for some TSM expression, ϵ.

A *peTSM definition context*, Ψ, is a finite function that maps each symbol $a \in \mathrm{dom}(\Psi)$ to a *peTSM definition*, $a \hookrightarrow \mathtt{petsm}(\rho; e_{\mathrm{parse}})$ for some TSM type ρ and expanded expression e_{parse}.

Similarly, a *ppTSM definition context*, Φ, is a finite function that maps each symbol $a \in \mathrm{dom}(\Phi)$ to a *ppTSM definition*, $a \hookrightarrow \mathtt{pptsm}(\rho; e_{\mathrm{parse}})$ for some TSM type ρ and expanded expression e_{parse}.

Kinds and Constructors

Kind expansion

$$\frac{\hat{\Omega} \vdash \hat{\kappa}_1 \rightsquigarrow \kappa_1 \ \mathsf{kind} \qquad \hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa_1 \vdash \hat{\kappa}_2 \rightsquigarrow \kappa_2 \ \mathsf{kind}}{\hat{\Omega} \vdash \mathtt{udarr}(\hat{\kappa}_1; \hat{u}.\hat{\kappa}_2) \rightsquigarrow \mathtt{darr}(\kappa_1; u.\kappa_2) \ \mathsf{kind}} \tag{5.1a}$$

$$\frac{}{\hat{\Omega} \vdash \mathtt{uunit} \rightsquigarrow \mathtt{unit} \ \mathsf{kind}} \tag{5.1b}$$

$$\frac{\hat{\Omega} \vdash \hat{\kappa}_1 \rightsquigarrow \kappa_1 \ \mathsf{kind} \qquad \hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa_1 \vdash \hat{\kappa}_2 \rightsquigarrow \kappa_2 \ \mathsf{kind}}{\hat{\Omega} \vdash \mathtt{udprod}(\hat{\kappa}_1; \hat{u}.\hat{\kappa}_2) \rightsquigarrow \mathtt{dprod}(\kappa_1; u.\kappa_2) \ \mathsf{kind}} \tag{5.1c}$$

$$\frac{}{\hat{\Omega} \vdash \mathtt{uType} \rightsquigarrow \mathtt{Type} \ \mathsf{kind}} \tag{5.1d}$$

$$\frac{\hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau \Leftarrow \mathtt{Type}}{\hat{\Omega} \vdash \mathtt{uS}(\hat{\tau}) \rightsquigarrow \mathtt{S}(\tau) \ \mathsf{kind}} \tag{5.1e}$$

Synthetic constructor expansion

$$\frac{}{\hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa \vdash \hat{u} \rightsquigarrow u \Rightarrow \kappa} \tag{5.2a}$$

$$\frac{\hat{\Omega} \vdash \hat{\kappa} \rightsquigarrow \kappa \ \mathsf{kind} \qquad \hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Leftarrow \kappa}{\hat{\Omega} \vdash \mathtt{uasc\{\hat{\kappa}\}(\hat{c})} \rightsquigarrow c \Rightarrow \kappa} \tag{5.2b}$$

$$\frac{\hat{\Omega} \vdash \hat{c}_1 \rightsquigarrow c_1 \Rightarrow \mathtt{darr}(\kappa_2; u.\kappa) \qquad \hat{\Omega} \vdash \hat{c}_2 \rightsquigarrow c_2 \Leftarrow \kappa_2}{\hat{\Omega} \vdash \mathtt{uapp}(\hat{c}_1; \hat{c}_2) \rightsquigarrow \mathtt{app}(c_1; c_2) \Rightarrow [c_1/u]\kappa} \tag{5.2c}$$

$$\frac{}{\hat{\Omega} \vdash \mathtt{utriv} \rightsquigarrow \mathtt{triv} \Rightarrow \mathtt{unit}} \tag{5.2d}$$

$$\frac{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Rightarrow \mathtt{dprod}(\kappa_1; u.\kappa_2)}{\hat{\Omega} \vdash \mathtt{uprl}(\hat{c}) \rightsquigarrow \mathtt{prl}(c) \Rightarrow \kappa_1} \tag{5.2e}$$

$$\frac{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Rightarrow \mathtt{dprod}(\kappa_1; u.\kappa_2)}{\hat{\Omega} \vdash \mathtt{uprr}(\hat{c}) \rightsquigarrow \mathtt{prr}(c) \Rightarrow [\mathtt{prl}(c)/u]\kappa_2} \tag{5.2f}$$

$$\frac{\hat{\Omega} \vdash \hat{\tau}_1 \rightsquigarrow \tau_1 \Leftarrow \mathtt{Type} \qquad \hat{\Omega} \vdash \hat{\tau}_2 \rightsquigarrow \tau_2 \Leftarrow \mathtt{Type}}{\hat{\Omega} \vdash \mathtt{uparr}(\hat{\tau}_1; \hat{\tau}_2) \rightsquigarrow \mathtt{parr}(\tau_1; \tau_2) \Rightarrow \mathtt{Type}} \tag{5.2g}$$

$$\frac{\hat{\Omega} \vdash \hat{\kappa} \rightsquigarrow \kappa \ \mathsf{kind} \qquad \hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa \vdash \hat{\tau} \rightsquigarrow \tau \Leftarrow \mathtt{Type}}{\hat{\Omega} \vdash \mathtt{uall\{\hat{\kappa}\}(\hat{u}.\hat{\tau})} \rightsquigarrow \mathtt{all\{\kappa\}(u.\tau)} \Rightarrow \mathtt{Type}} \tag{5.2h}$$

$$\frac{\hat{\Omega}, \hat{t} \rightsquigarrow t :: \mathtt{Type} \vdash \hat{\tau} \rightsquigarrow \tau \Leftarrow \mathtt{Type}}{\hat{\Omega} \vdash \mathtt{urec}(\hat{t}.\hat{\tau}) \rightsquigarrow \mathtt{rec}(t.\tau) \Rightarrow \mathtt{Type}} \tag{5.2i}$$

$$\frac{\{\hat{\Omega} \vdash \hat{\tau}_i \rightsquigarrow \tau_i \Leftarrow \mathtt{Type}\}_{1 \leq i \leq n}}{\hat{\Omega} \vdash \mathtt{uprod}[L](\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}) \rightsquigarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \Rightarrow \mathtt{Type}} \tag{5.2j}$$

$$\frac{\{\hat{\Omega} \vdash \hat{\tau}_i \rightsquigarrow \tau_i \Leftarrow \mathtt{Type}\}_{1 \leq i \leq n}}{\hat{\Omega} \vdash \mathtt{usum}[L](\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}) \rightsquigarrow \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \Rightarrow \mathtt{Type}} \tag{5.2k}$$

$$\frac{}{\hat{\Omega}, \hat{X} \rightsquigarrow X : \mathtt{sig\{\kappa\}(u.\tau)} \vdash \mathtt{ucon}(\hat{X}) \rightsquigarrow \mathtt{con}(X) \Rightarrow \kappa} \tag{5.2l}$$

Analytic constructor expansion

$$\frac{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Rightarrow \kappa_1 \qquad \Omega \vdash \kappa_1 <:: \kappa_2}{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Leftarrow \kappa_2} \tag{5.3a}$$

$$\frac{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Leftarrow \mathtt{Type}}{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Leftarrow \mathtt{S}(c)} \tag{5.3b}$$

$$\frac{\hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa_1 \vdash \hat{c}_2 \rightsquigarrow c_2 \Leftarrow \kappa_2}{\hat{\Omega} \vdash \mathtt{uabs}(\hat{u}.\hat{c}_2) \rightsquigarrow \mathtt{abs}(u.c_2) \Leftarrow \mathtt{darr}(\kappa_1; u.\kappa_2)} \tag{5.3c}$$

$$\frac{\hat{\Omega} \vdash \hat{c}_1 \rightsquigarrow c_1 \Leftarrow \kappa_1 \qquad \hat{\Omega} \vdash \hat{c}_2 \rightsquigarrow c_2 \Leftarrow [c_1/u]\kappa_2}{\hat{\Omega} \vdash \mathtt{upair}(\hat{c}_1; \hat{c}_2) \rightsquigarrow \mathtt{pair}(c_1; c_2) \Leftarrow \mathtt{dprod}(\kappa_1; u.\kappa_2)} \tag{5.3d}$$

Types, Expressions, Rules and Patterns

Type expansion

$$\frac{\hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau \Leftarrow \mathsf{Type}}{\hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type}} \tag{5.4}$$

Synthetic typed expression expansion

$$\frac{}{\Omega, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{x} \rightsquigarrow x \Rightarrow \tau} \tag{5.5a}$$

$$\frac{\hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uasc\{}\hat{\tau}\mathtt{\}(}\hat{e}\mathtt{)} \rightsquigarrow e \Rightarrow \tau} \tag{5.5b}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \hat{\Omega}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \rightsquigarrow e' \Rightarrow \tau'}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uletval(}\hat{e}\mathtt{;}\hat{x}.\hat{e}'\mathtt{)} \rightsquigarrow \mathtt{ap(lam\{}\tau\mathtt{\}(}x.e'\mathtt{);}e\mathtt{)} \Rightarrow \tau'} \tag{5.5c}$$

$$\frac{\hat{\Omega} \vdash \hat{\tau}_1 \rightsquigarrow \tau_1\ \mathsf{type} \qquad \hat{\Omega}, \hat{x} \rightsquigarrow x : \tau_1 \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau_2}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{ulam\{}\hat{\tau}_1\mathtt{\}(}\hat{x}.\hat{e}\mathtt{)} \rightsquigarrow \mathtt{lam\{}\tau_1\mathtt{\}(}x.e\mathtt{)} \Rightarrow \mathtt{parr(}\tau_1\mathtt{;}\tau_2\mathtt{)}} \tag{5.5d}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_1 \rightsquigarrow e_1 \Rightarrow \mathtt{parr(}\tau_2\mathtt{;}\tau\mathtt{)} \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_2 \rightsquigarrow e_2 \Leftarrow \tau_2}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uap(}\hat{e}_1\mathtt{;}\hat{e}_2\mathtt{)} \rightsquigarrow \mathtt{ap(}e_1\mathtt{;}e_2\mathtt{)} \Rightarrow \tau} \tag{5.5e}$$

$$\frac{\hat{\Omega} \vdash \hat{\kappa} \rightsquigarrow \kappa\ \mathsf{kind} \qquad \hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uclam\{}\hat{\kappa}\mathtt{\}(}\hat{u}.\hat{e}\mathtt{)} \rightsquigarrow \mathtt{clam\{}\kappa\mathtt{\}(}u.e\mathtt{)} \Rightarrow \mathtt{all\{}\kappa\mathtt{\}(}u.\tau\mathtt{)}} \tag{5.5f}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \mathtt{all\{}\kappa\mathtt{\}(}u.\tau\mathtt{)} \qquad \hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Rightarrow \kappa}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{ucap\{}\hat{c}\mathtt{\}(}\hat{e}\mathtt{)} \rightsquigarrow \mathtt{cap\{}c\mathtt{\}(}e\mathtt{)} \Rightarrow [c/t]\tau} \tag{5.5g}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \mathtt{rec(}t.\tau\mathtt{)}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uunfold(}\hat{e}\mathtt{)} \rightsquigarrow \mathtt{unfold(}e\mathtt{)} \Rightarrow [\mathtt{rec(}t.\tau\mathtt{)}/t]\tau} \tag{5.5h}$$

$$\frac{\hat{e} = \mathtt{utpl\{}L\mathtt{\}(}\{i \hookrightarrow \hat{e}_i\}_{i \in L}\mathtt{)} \qquad e = \mathtt{tpl}[L]\mathtt{(}\{i \hookrightarrow e_i\}_{i \in L}\mathtt{)} \qquad \{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_i \rightsquigarrow e_i \Rightarrow \tau_i\}_{i \in L}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \mathtt{prod}[L]\mathtt{(}\{i \hookrightarrow \tau_i\}_{i \in L}\mathtt{)}} \tag{5.5i}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \mathtt{prod}[L, \ell]\mathtt{(}\{i \hookrightarrow \tau_i\}_{i \in L}\mathtt{;}\ell \hookrightarrow \tau\mathtt{)}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uprj}[\ell]\mathtt{(}\hat{e}\mathtt{)} \rightsquigarrow \mathtt{prj}[\ell]\mathtt{(}e\mathtt{)} \Rightarrow \tau} \tag{5.5j}$$

$$\frac{n > 0 \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \rightsquigarrow r_i \Rightarrow \tau \Rightarrow \tau'\}_{1 \leq i \leq n}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{umatch}[n]\mathtt{(}\hat{e}\mathtt{;}\{\hat{r}_i\}_{1 \leq i \leq n}\mathtt{)} \rightsquigarrow \mathtt{match}[n]\mathtt{\{}\tau'\mathtt{\}(}e\mathtt{;}\{r_i\}_{1 \leq i \leq n}\mathtt{)} \Rightarrow \tau'} \tag{5.5k}$$

$$\frac{}{\hat{\Omega}, \hat{X} \rightsquigarrow X : \mathtt{sig\{}\kappa\mathtt{\}(}u.\tau\mathtt{)} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uval(}\hat{X}\mathtt{)} \rightsquigarrow \mathtt{val(}X\mathtt{)} \Rightarrow [\mathtt{con(}X\mathtt{)}/u]\tau} \tag{5.5l}$$

$$\frac{\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho\ \mathsf{tsmty} \qquad \emptyset \vdash e_{\mathsf{parse}} : \mathtt{parr}(\mathtt{Body};\mathtt{ParseResult}(\mathtt{PCEExp})) \qquad \hat{\Omega} \vdash_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \mathtt{defref}[a];\Psi, a \hookrightarrow \mathtt{petsm}(\rho;e_{\mathsf{parse}});\mathcal{I}\rangle;\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau}{\hat{\Omega} \vdash_{\langle \mathcal{A};\Psi;\mathcal{I}\rangle;\hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for}\ \mathtt{expressions}\ \{e_{\mathsf{parse}}\}\ \mathtt{in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau} \tag{5.5m}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\langle \mathcal{A};\Psi;\mathcal{I}\rangle} \hat{e} \rightsquigarrow \epsilon\ @\ \rho \qquad \hat{\Omega} \vdash_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \epsilon;\Psi;\mathcal{I}\rangle;\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau}{\hat{\Omega} \vdash_{\langle \mathcal{A};\Psi;\mathcal{I}\rangle;\hat{\Phi}} \mathtt{let\ syntax}\ \hat{a} = \hat{e}\ \mathtt{for\ expressions\ in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau} \tag{5.5n}$$

$$\frac{\begin{array}{c}\hat{\Omega} = \langle \mathcal{D};\mathcal{G};\mathcal{M};\Omega_{\mathsf{app}}\rangle \qquad \hat{\Psi} = \langle \mathcal{A};\Psi, a \hookrightarrow \mathtt{petsm}(\rho;e_{\mathsf{parse}});\mathcal{I}\rangle \\ \hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{e} \rightsquigarrow \epsilon\ @\ \mathtt{type}(\tau_{\mathsf{final}}) \qquad \Omega_{\mathsf{app}} \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \Downarrow \epsilon_{\mathsf{normal}} \\ \mathsf{tsmdef}(\epsilon_{\mathsf{normal}}) = a \qquad b \downarrow_{\mathsf{Body}} e_{\mathsf{body}} \qquad e_{\mathsf{parse}}(e_{\mathsf{body}}) \Downarrow \mathtt{inj}[\mathtt{Success}](e_{\mathsf{pcand}}) \\ e_{\mathsf{pcand}} \uparrow_{\mathsf{PCEExp}} \check{e} \\ \Omega_{\mathsf{app}} \vdash^{\mathsf{Exp}}_{\Psi, a \hookrightarrow \mathtt{petsm}(\rho;e_{\mathsf{parse}})} \check{e} \curvearrowright_{\epsilon_{\mathsf{normal}}} \grave{e}\ ?\ \mathtt{type}(\tau_{\mathsf{cand}}) \dashv \omega : \Omega_{\mathsf{params}} \\ \Omega_{\mathsf{params}} \vdash^{\hat{\Omega};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e \Leftarrow \tau_{\mathsf{cand}}\end{array}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}\ /b/ \rightsquigarrow [\omega]e \Rightarrow [\omega]\tau_{\mathsf{cand}}} \tag{5.5o}$$

The basic idea here is that we need to:

1. Expand the TSM expression ϵ.
2. Normalize the expansion of the TSM expression.
3. Extract the TSM definition at the head, a.
4. Call the corresponding parse function. The result is a parameterized candidate expansion expression, $\check{e}$.
5. We then need to unwrap the candidate expansion under the binders, producing a substitution ω and a corresponding context, Ω_{params}. This is necessary so that alpha-renaming works properly.
6. Finally, we validate the unwrapped candidate expansion.

TODO: peTSM implicit designation

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a;\Psi, a \hookrightarrow \mathtt{setsm}(\tau;e_{\mathsf{parse}});\mathcal{I} \uplus \tau \hookrightarrow a\rangle;\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a;\Psi, a \hookrightarrow \mathtt{setsm}(\tau;e_{\mathsf{parse}});\mathcal{I}\rangle;\hat{\Phi}} \mathtt{implicit\ syntax}\ \hat{a}\ \mathtt{for\ expressions\ in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \tag{5.5p}$$

$$\frac{\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho\ \mathsf{tsmty} \qquad \emptyset \vdash e_{\mathsf{parse}} : \mathtt{parr}(\mathtt{Body};\mathtt{ParseResult}(\mathtt{prPat})) \qquad \hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \mathtt{defref}[a];\Phi, a \hookrightarrow \mathtt{pptsm}(\rho;e_{\mathsf{parse}});\mathcal{I}\rangle} \hat{e} \rightsquigarrow e \Rightarrow \tau}{\hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A};\Phi;\mathcal{I}\rangle} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for\ patterns\ by\ static}\ e_{\mathsf{parse}}\ \mathtt{end\ in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau} \tag{5.5q}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\langle \mathcal{A};\Phi;\mathcal{I}\rangle} \hat{e} \rightsquigarrow \epsilon\ @\ \rho \qquad \hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \epsilon;\Phi;\mathcal{I}\rangle} \hat{e} \rightsquigarrow e \Rightarrow \tau}{\hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A};\Phi;\mathcal{I}\rangle} \mathtt{let\ syntax}\ \hat{a} = \hat{e}\ \mathtt{for\ patterns\ in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau} \tag{5.5r}$$

TODO: ppTSM implicit designation

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \mathtt{sptsm}(\tau; e_{\mathrm{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle} \hat{e} \rightsquigarrow e \Rightarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \mathtt{setsm}(\tau; e_{\mathrm{parse}}); \mathcal{I} \rangle} \mathtt{implicit\ syntax}\ \hat{a}\ \mathtt{for\ patterns\ in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \tag{5.5s}$$

Analytic typed expression expansion

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \Omega \vdash \tau <: \tau'}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{5.6a}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \hat{\Omega}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \rightsquigarrow e' \Leftarrow \tau'}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uletval}(\hat{e}; \hat{x}.\hat{e}') \rightsquigarrow \mathtt{ap}(\mathtt{lam}\{\tau\}(x.e'); e) \Leftarrow \tau'} \tag{5.6b}$$

$$\frac{\hat{\Omega}, \hat{x} \rightsquigarrow x : \tau_1 \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau_2}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uanalam}(\hat{x}.\hat{e}) \rightsquigarrow \mathtt{lam}\{\tau_1\}(x.e) \Leftarrow \mathtt{parr}(\tau_1; \tau_2)} \tag{5.6c}$$

$$\frac{\hat{\Delta}, \hat{t} \rightsquigarrow t\ \mathsf{type}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{utlam}(\hat{t}.\hat{e}) \rightsquigarrow \mathtt{tlam}(t.e) \Leftarrow \mathtt{all}(t.\tau)} \tag{5.6d}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow [\mathtt{rec}(t.\tau)/t]\tau}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{ufold}(\hat{e}) \rightsquigarrow \mathtt{fold}\{t.\tau\}(e) \Leftarrow \mathtt{rec}(t.\tau)} \tag{5.6e}$$

$$\frac{\hat{e} = \mathtt{utpl}\{L\}(\{i \hookrightarrow \hat{e}_i\}_{i \in L}) \qquad e = \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) \qquad \{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_i \rightsquigarrow e_i \Leftarrow \tau_i\}_{i \in L}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{5.6f}$$

$$\frac{\hat{e} = \mathtt{uin}[\ell](\hat{e}') \qquad e = \mathtt{inj}[L, \ell; \ell]\{\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau\}(e') \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \rightsquigarrow e' \Leftarrow \tau}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)} \tag{5.6g}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \rightsquigarrow r_i \Leftarrow \tau \Rightarrow \tau'\}_{1 \le i \le n}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{umatch}[n](\hat{e}; \{\hat{r}_i\}_{1 \le i \le n}) \rightsquigarrow \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \le i \le n}) \Leftarrow \tau'} \tag{5.6h}$$

$$\frac{\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho\ \mathsf{tsmty} \qquad \emptyset \vdash e_{\mathrm{parse}} : \mathtt{parr}(\mathtt{Body}; \mathtt{ParseResult}(\mathtt{PCEExp})) \qquad \hat{\Omega} \vdash_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \mathtt{defref}[a]; \Psi, a \hookrightarrow \mathtt{petsm}(\rho; e_{\mathrm{parse}}); \mathcal{I} \rangle; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Omega} \vdash_{\langle \mathcal{A}; \Psi; \mathcal{I} \rangle; \hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for\ \ expressions}\ \{e_{\mathrm{parse}}\}\ \mathtt{in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau} \tag{5.6i}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\langle \mathcal{A}; \Psi; \mathcal{I} \rangle} \hat{\epsilon} \rightsquigarrow \epsilon\ @\ \rho \qquad \hat{\Omega} \vdash_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \epsilon; \Psi; \mathcal{I} \rangle; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Omega} \vdash_{\langle \mathcal{A}; \Psi; \mathcal{I} \rangle; \hat{\Phi}} \mathtt{let\ syntax}\ \hat{a} = \hat{\epsilon}\ \mathtt{for\ expressions\ in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau} \tag{5.6j}$$

TODO: peTSM implicit designation

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \mathtt{setsm}(\tau; e_{\mathrm{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \mathtt{setsm}(\tau; e_{\mathrm{parse}}); \mathcal{I} \rangle; \hat{\Phi}} \mathtt{implicit\ syntax}\ \hat{a}\ \mathtt{for\ expressions\ in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{5.6k}$$

TODO: peTSM implicit application

$$\frac{b \downarrow_{\mathsf{Body}} e_{\mathrm{body}} \qquad e_{\mathrm{parse}}(e_{\mathrm{body}}) \Downarrow \mathtt{inj}[\mathtt{Success}](e_{\mathrm{proto}}) \qquad e_{\mathrm{proto}} \uparrow_{\mathsf{ProtoExpr}} \grave{e} \\ \varnothing\ \varnothing \vdash^{\hat{\Delta}; \hat{\Gamma}; \langle \mathcal{A}; \Psi, a \hookrightarrow \mathtt{setsm}(\tau; e_{\mathrm{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle; \hat{\Phi}; b} \grave{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A}; \Psi, a \hookrightarrow \mathtt{setsm}(\tau; e_{\mathrm{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle; \hat{\Phi}} \mathtt{uelit}[b] \rightsquigarrow e \Leftarrow \tau} \tag{5.6l}$$

$$\frac{\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho\ \mathsf{tsmty} \qquad \varnothing \vdash e_{\mathrm{parse}} : \mathtt{parr}(\mathtt{Body}; \mathtt{ParseResult}(\mathtt{prPat})) \\ \hat{\Omega} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \hookrightarrow \mathtt{defref}[a]; \Phi, a \hookrightarrow \mathtt{pptsm}(\rho; e_{\mathrm{parse}}); \mathcal{I} \rangle} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Omega} \vdash_{\hat{\Psi}; \langle \mathcal{A}; \Phi; \mathcal{I} \rangle} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for\ patterns\ by\ static}\ e_{\mathrm{parse}}\ \mathtt{end\ in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau} \tag{5.6m}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\langle \mathcal{A}; \Phi; \mathcal{I} \rangle} \hat{\epsilon} \rightsquigarrow \epsilon\ @\ \rho \qquad \hat{\Omega} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \hookrightarrow \epsilon; \Phi; \mathcal{I} \rangle} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Omega} \vdash_{\hat{\Psi}; \langle \mathcal{A}; \Phi; \mathcal{I} \rangle} \mathtt{let\ syntax}\ \hat{a} = \hat{\epsilon}\ \mathtt{for\ patterns\ in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau} \tag{5.6n}$$

TODO: ppTSM implicit designation

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \mathtt{sptsm}(\tau; e_{\mathrm{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \mathtt{setsm}(\tau; e_{\mathrm{parse}}); \mathcal{I} \rangle} \mathtt{implicit\ syntax}\ \hat{a}\ \mathtt{for\ patterns\ in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{5.6o}$$

Synthetic rule expansion

$$\frac{\hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega \rangle \\ \hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \langle \varnothing; \mathcal{G}'; \varnothing; \Omega' \rangle \qquad \langle \mathcal{D}; \mathcal{G} \uplus \mathcal{G}'; \mathcal{M}; \Omega \cup \Omega' \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau'}{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \mathtt{urule}(\hat{p}.\hat{e}) \rightsquigarrow \mathtt{rule}(p.e) \Rightarrow \tau \mapsto \tau'} \tag{5.7}$$

Analytic rule expansion

$$\frac{\hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega \rangle \\ \hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \langle \varnothing; \mathcal{G}'; \varnothing; \Omega' \rangle \qquad \langle \mathcal{D}; \mathcal{G} \uplus \mathcal{G}'; \mathcal{M}; \Omega \cup \Omega' \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \mathtt{urule}(\hat{p}.\hat{e}) \rightsquigarrow \mathtt{rule}(p.e) \Leftarrow \tau \mapsto \tau'} \tag{5.8}$$

Typed pattern expansion

$$\frac{\hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega \rangle \\ \hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \hat{\Omega}' \qquad \Omega \vdash \tau <: \tau'}{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau' \dashv\!\mid \hat{\Omega}'} \tag{5.9a}$$

$$\frac{}{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{x} \rightsquigarrow x : \tau \dashv\!\shortmid \langle \emptyset; \hat{x} \rightsquigarrow x; \emptyset; x : \tau \rangle} \tag{5.9b}$$

$$\frac{}{\hat{\Omega} \vdash_{\hat{\Phi}} \texttt{uwildp} \rightsquigarrow \texttt{wildp} : \tau \dashv\!\shortmid \langle \emptyset; \emptyset; \emptyset; \emptyset \rangle} \tag{5.9c}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : [\texttt{rec}(t.\tau)/t]\tau \dashv\!\shortmid \hat{\Omega}'}{\hat{\Omega} \vdash_{\hat{\Phi}} \texttt{ufoldp}(\hat{p}) \rightsquigarrow \texttt{foldp}(p) : \texttt{rec}(t.\tau) \dashv\!\shortmid \hat{\Omega}'} \tag{5.9d}$$

$$\frac{\hat{p} = \texttt{utplp}[L](\{i \hookrightarrow \hat{p}_i\}_{i \in L}) \qquad p = \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) \qquad \{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{p}_i \rightsquigarrow p_i : \tau_i \dashv\!\shortmid \hat{\Omega}_i\}_{i \in L}}{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\!\shortmid \cup_{i \in L} \hat{\Omega}_i} \tag{5.9e}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\shortmid \hat{\Omega}'}{\hat{\Omega} \vdash_{\hat{\Phi}} \texttt{uinjp}[\ell](\hat{p}) \rightsquigarrow \texttt{injp}[\ell](p) : \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\!\shortmid \hat{\Omega}'} \tag{5.9f}$$

$$\frac{\begin{array}{c}\hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega_{\text{app}} \rangle \qquad \hat{\Phi} = \langle \mathcal{A}; \Phi, a \hookrightarrow \texttt{pptsm}(\rho; e_{\text{parse}}); \mathcal{I} \rangle \\ \hat{\Omega} \vdash^{\text{Pat}}_{\hat{\Phi}} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \texttt{type}(\tau_{\text{final}}) \qquad \text{tsmdef}(\epsilon) = a \\ b \downarrow_{\text{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}[\texttt{Success}](e_{\text{proto}}) \qquad e_{\text{proto}} \uparrow_{\text{ProtoPat}} \grave{p} \\ \Omega_{\text{app}} \vdash^{\text{Pat}}_{\Phi, a \hookrightarrow \texttt{pptsm}(\rho; e_{\text{parse}})} \cdot \looparrowright_{\epsilon} \cdot \,?\, \texttt{type}(\tau_{\text{cand}}) \dashv \omega : \Omega_{\text{params}} \\ \Omega_{\text{param}} \vdash^{\hat{\Omega}; \hat{\Phi}; b} \grave{p} \rightsquigarrow p : \tau_{\text{cand}} \dashv\!\shortmid^{\hat{\Omega}'}\end{array}}{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{e}\ /b/ \rightsquigarrow p : [\omega]\tau_{\text{cand}} \dashv\!\shortmid \hat{\Omega}'} \tag{5.9g}$$

TODO: ppTSM implicit application

$$\frac{\begin{array}{c}b \downarrow_{\text{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}[\texttt{Success}](e_{\text{proto}}) \qquad e_{\text{proto}} \uparrow_{\text{ProtoPat}} \grave{p} \\ \Omega_{\text{param}} \vdash^{\Delta; \langle \mathcal{A}; \Phi, a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}}); \mathcal{I}, \tau \hookrightarrow a \rangle; b} \grave{p} \rightsquigarrow p : \tau \dashv\!\shortmid^{\hat{\Omega}}\end{array}}{\Delta \vdash_{\langle \mathcal{A}; \Phi, a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}}); \mathcal{I}, \tau \hookrightarrow a \rangle} /b/ \rightsquigarrow p : \tau \dashv\!\shortmid \hat{\Omega}} \tag{5.9h}$$

Unexpanded Signatures and Module Expressions

Signature expansion

$$\frac{\hat{\Omega} \vdash \hat{\kappa} \rightsquigarrow \kappa \text{ kind} \qquad \hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa \vdash \hat{\tau} \rightsquigarrow \tau \text{ type}}{\hat{\Omega} \vdash \texttt{usig}\{\hat{\kappa}\}(\hat{u}.\hat{\tau}) \rightsquigarrow \texttt{sig}\{\kappa\}(u.\tau) \text{ sig}} \tag{5.10}$$

Synthetic module expression expansion

$$\frac{}{\hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma \vdash_{\hat{\Phi}; \hat{\Psi}} \hat{X} \rightsquigarrow X \Rightarrow \sigma} \tag{5.11a}$$

$$\frac{\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma \text{ sig} \qquad \hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{M} \rightsquigarrow M \Leftarrow \sigma}{\hat{\Omega} \vdash_{\hat{\Psi}_{\text{E}}; \hat{\Psi}_{\text{P}}} \texttt{useal}\{\hat{\sigma}\}(\hat{M}) \rightsquigarrow \texttt{seal}\{\sigma\}(M) \Rightarrow \sigma} \tag{5.11b}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi}_E;\hat{\Psi}_P} \hat{M} \rightsquigarrow M \Rightarrow \sigma \qquad \hat{\Omega} \vdash \hat{\sigma}' \rightsquigarrow \sigma' \ \mathsf{sig} \quad \hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma \vdash_{\hat{\Psi};\hat{\Phi}} \hat{M}' \rightsquigarrow M' \Leftarrow \sigma'}{\hat{\Omega} \vdash_{\hat{\Psi}_E;\hat{\Psi}_P} \mathtt{umlet\{}\hat{\sigma}'\mathtt{\}(}\hat{M}; \hat{X}.\hat{M}'\mathtt{)} \rightsquigarrow \mathtt{mlet\{}\sigma'\mathtt{\}(}M; X.M'\mathtt{)} \Rightarrow \sigma'} \tag{5.11c}$$

$$\frac{\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho \ \mathsf{tsmty} \qquad \varnothing \vdash e_{\mathrm{parse}} : \mathtt{parr(Body;ParseResult(PCEExp))} \quad \hat{\Omega} \vdash_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \mathtt{defref[}a\mathtt{]};\Psi, a \hookrightarrow \mathtt{petsm}(\rho; e_{\mathrm{parse}});\mathcal{I}\rangle;\hat{\Phi}} \hat{M} \rightsquigarrow M \Rightarrow \sigma}{\hat{\Omega} \vdash_{\langle \mathcal{A};\Psi;\mathcal{I}\rangle;\hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for\ \ expressions}\ \{e_{\mathrm{parse}}\}\ \mathtt{in}\ \hat{M} \rightsquigarrow M \Rightarrow \sigma} \tag{5.11d}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\langle \mathcal{A};\Psi;\mathcal{I}\rangle} \hat{e} \rightsquigarrow \epsilon \mathbin{@} \rho \qquad \hat{\Omega} \vdash_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \epsilon;\Psi;\mathcal{I}\rangle;\hat{\Phi}} \hat{M} \rightsquigarrow M \Rightarrow \sigma}{\hat{\Omega} \vdash_{\langle \mathcal{A};\Psi;\mathcal{I}\rangle;\hat{\Phi}} \mathtt{let\ syntax}\ \hat{a} = \hat{e}\ \mathtt{for\ expressions\ in}\ \hat{M} \rightsquigarrow M \Rightarrow \sigma} \tag{5.11e}$$

TODO: peTSM implicit designation at module level

$$\frac{\cdots}{\cdots} \tag{5.11f}$$

$$\frac{\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho \ \mathsf{tsmty} \qquad \varnothing \vdash e_{\mathrm{parse}} : \mathtt{parr(Body;ParseResult(prPat))} \quad \hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \mathtt{defref[}a\mathtt{]};\Phi, a \hookrightarrow \mathtt{pptsm}(\rho; e_{\mathrm{parse}});\mathcal{I}\rangle} \hat{M} \rightsquigarrow M \Rightarrow \sigma}{\hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A};\Phi;\mathcal{I}\rangle} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for\ patterns\ by\ static}\ e_{\mathrm{parse}}\ \mathtt{end\ in}\ \hat{M} \rightsquigarrow M \Rightarrow \sigma} \tag{5.11g}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\langle \mathcal{A};\Phi;\mathcal{I}\rangle} \hat{e} \rightsquigarrow \epsilon \mathbin{@} \rho \qquad \hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \epsilon;\Phi;\mathcal{I}\rangle} \hat{M} \rightsquigarrow M \Rightarrow \sigma}{\hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A};\Phi;\mathcal{I}\rangle} \mathtt{let\ syntax}\ \hat{a} = \hat{e}\ \mathtt{for\ patterns\ in}\ \hat{M} \rightsquigarrow M \Rightarrow \sigma} \tag{5.11h}$$

TODO: ppTSM implicit designation at module level

$$\frac{\cdots}{\cdots} \tag{5.11i}$$

Analytic module expression expansion

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi}_E;\hat{\Psi}_P} \hat{M} \rightsquigarrow M \Rightarrow \sigma \qquad \hat{\Omega} \vdash \sigma <: \sigma'}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{M} \rightsquigarrow M \Leftarrow \sigma'} \tag{5.12a}$$

$$\frac{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Leftarrow \kappa \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow [c/u]\tau}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{ustruct(}\hat{c};\hat{e}\mathtt{)} \rightsquigarrow \mathtt{struct(}c;e\mathtt{)} \Leftarrow \mathtt{sig\{}\kappa\mathtt{\}(}u.\tau\mathtt{)}} \tag{5.12b}$$

$$\frac{\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho \ \mathsf{tsmty} \qquad \varnothing \vdash e_{\mathrm{parse}} : \mathtt{parr(Body;ParseResult(PCEExp))} \quad \hat{\Omega} \vdash_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \mathtt{defref[}a\mathtt{]};\Psi, a \hookrightarrow \mathtt{petsm}(\rho; e_{\mathrm{parse}});\mathcal{I}\rangle;\hat{\Phi}} \hat{M} \rightsquigarrow M \Leftarrow \sigma}{\hat{\Omega} \vdash_{\langle \mathcal{A};\Psi;\mathcal{I}\rangle;\hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for\ \ expressions}\ \{e_{\mathrm{parse}}\}\ \mathtt{in}\ \hat{M} \rightsquigarrow M \Leftarrow \sigma} \tag{5.12c}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\langle \mathcal{A};\Psi;\mathcal{I}\rangle} \hat{e} \rightsquigarrow e \mathbin{@} \rho \qquad \hat{\Omega} \vdash_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow e;\Psi;\mathcal{I}\rangle;\hat{\Phi}} \hat{M} \rightsquigarrow M \Leftarrow \sigma}{\hat{\Omega} \vdash_{\langle \mathcal{A};\Psi;\mathcal{I}\rangle;\hat{\Phi}} \texttt{let syntax } \hat{a} = \hat{e} \texttt{ for expressions in } \hat{M} \rightsquigarrow M \Leftarrow \sigma} \tag{5.12d}$$

TODO: peTSM implicit designation at module level

$$\frac{\cdots}{\cdots} \tag{5.12e}$$

$$\frac{\begin{array}{c}\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho \text{ tsmty} \qquad \emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResult(prPat))} \\ \hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \texttt{defref[}a\texttt{]};\Phi,a \hookrightarrow \texttt{pptsm}(\rho;e_{\text{parse}});\mathcal{I}\rangle} \hat{M} \rightsquigarrow M \Leftarrow \sigma\end{array}}{\hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A};\Phi;\mathcal{I}\rangle} \texttt{syntax } \hat{a} \texttt{ at } \hat{\rho} \texttt{ for patterns by static } e_{\text{parse}} \texttt{ end in } \hat{M} \rightsquigarrow M \Leftarrow \sigma} \tag{5.12f}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\langle \mathcal{A};\Phi;\mathcal{I}\rangle} \hat{e} \rightsquigarrow e \mathbin{@} \rho \qquad \hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \hookrightarrow e;\Phi;\mathcal{I}\rangle} \hat{M} \rightsquigarrow M \Leftarrow \sigma}{\hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A};\Phi;\mathcal{I}\rangle} \texttt{let syntax } \hat{a} = \hat{e} \texttt{ for patterns in } \hat{M} \rightsquigarrow M \Leftarrow \sigma} \tag{5.12g}$$

TODO: ppTSM implicit designation at module level

$$\frac{\cdots}{\cdots} \tag{5.12h}$$

TSM Types and Expressions

TSM Expression Typing

Judgement Form	**Description**
$\Omega \vdash \rho$ tsmty	ρ is a well-formed TSM type
$\Omega \vdash^{\mathsf{Exp}}_{\Psi} e \mathbin{@} \rho$	peTSM expression e has TSM type ρ
$\Omega \vdash^{\mathsf{Pat}}_{\Phi} e \mathbin{@} \rho$	ppTSM expression e has TSM type ρ

peTSM Expression Evaluation

Judgement Form	**Description**
$\Omega \vdash^{\mathsf{Exp}}_{\Psi} e$ normal	peTSM expression e is in normal form
$\Omega \vdash^{\mathsf{Exp}}_{\Psi} e \mapsto e'$	peTSM expression e transitions to e'

+ auxiliary judgements for multi-step transitions and evaluation
unexpanded TSM types and expressions

Judgement Form	**Description**
$\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho$ tsmty	$\hat{\rho}$ has expansion ρ
$\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{e} \rightsquigarrow e \mathbin{@} \rho$	unexpanded peTSM expression $\hat{e}$ has expansion e and type ρ
$\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \hat{e} \rightsquigarrow e \mathbin{@} \rho$	unexpanded ppTSM expression $\hat{e}$ has expansion e and type ρ

TSM type formation

$$\frac{\Omega \vdash \tau \ \mathsf{type}}{\Omega \vdash \mathtt{type}(\tau)\ \mathsf{tsmty}} \tag{5.13a}$$

$$\frac{\Omega, t :: \mathsf{Type} \vdash \rho \ \mathsf{tsmty}}{\Omega \vdash \mathtt{alltypes}(t.\rho)\ \mathsf{tsmty}} \tag{5.13b}$$

$$\frac{\Omega \vdash \sigma \ \mathsf{sig} \qquad \Omega, X : \sigma \vdash \rho \ \mathsf{tsmty}}{\Omega \vdash \mathtt{allmods}\{\sigma\}(X.\rho)\ \mathsf{tsmty}} \tag{5.13c}$$

Unexpanded TSM type expansion

$$\frac{\hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau \ \mathsf{type}}{\hat{\Omega} \vdash \mathtt{utype}(\hat{\tau}) \rightsquigarrow \mathtt{type}(\tau)\ \mathsf{tsmty}} \tag{5.14a}$$

$$\frac{\hat{\Omega}, \hat{t} \rightsquigarrow t :: \mathsf{Type} \vdash \hat{\rho} \rightsquigarrow \rho \ \mathsf{tsmty}}{\hat{\Omega} \vdash \mathtt{ualltypes}(\hat{t}.\hat{\rho}) \rightsquigarrow \mathtt{alltypes}(t.\rho)\ \mathsf{tsmty}} \tag{5.14b}$$

$$\frac{\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma \ \mathsf{sig} \qquad \hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma \vdash \hat{\rho} \rightsquigarrow \rho \ \mathsf{tsmty}}{\hat{\Omega} \vdash \mathtt{uallmods}\{\hat{\sigma}\}(\hat{X}.\hat{\rho}) \rightsquigarrow \mathtt{allmods}\{\sigma\}(X.\rho)\ \mathsf{tsmty}} \tag{5.14c}$$

peTSM Expression Typing

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi, a \hookrightarrow \mathtt{petsm}(\rho; e_{\mathrm{parse}})} \mathtt{defref}[a] \mathbin{@} \rho} \tag{5.15a}$$

$$\frac{\Omega, t :: \mathsf{Type} \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} \rho}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{abstype}(t.\epsilon) \mathbin{@} \mathtt{alltypes}(t.\rho)} \tag{5.15b}$$

$$\frac{\Omega \vdash \sigma \ \mathsf{sig} \qquad \Omega, X : \sigma \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} \rho}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{absmod}\{\sigma\}(X.\epsilon) \mathbin{@} \mathtt{allmods}\{\sigma\}(X.\rho)} \tag{5.15c}$$

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} \mathtt{alltypes}(t.\rho) \qquad \Omega \vdash \tau \ \mathsf{type}}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{aptype}\{\tau\}(\epsilon) \mathbin{@} [\tau/t]\rho} \tag{5.15d}$$

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} \mathtt{allmods}\{\sigma\}(X'.\rho) \qquad \Omega \vdash X : \sigma}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{apmod}\{X\}(\epsilon) \mathbin{@} [X/X']\rho} \tag{5.15e}$$

ppTSM Expression Typing

$$\frac{}{\Omega \vdash^{\mathsf{Pat}}_{\Phi, a \hookrightarrow \mathtt{pptsm}(\rho; e_{\mathrm{parse}})} \mathtt{defref}[a] \mathbin{@} \rho} \tag{5.16a}$$

$$\frac{\Omega, t :: \mathsf{Type} \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mathbin{@} \rho}{\Omega \vdash^{\mathsf{Pat}}_{\Phi} \mathtt{abstype}(t.\epsilon) \mathbin{@} \mathtt{alltypes}(t.\rho)} \tag{5.16b}$$

$$\frac{\Omega \vdash \sigma\ \mathsf{sig} \qquad \Omega, X : \sigma \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mathbin{@} \rho}{\Omega \vdash^{\mathsf{Pat}}_{\Phi} \mathtt{absmod}\{\sigma\}(X.\epsilon) \mathbin{@} \mathtt{allmods}\{\sigma\}(X.\rho)} \tag{5.16c}$$

$$\frac{\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mathbin{@} \mathtt{alltypes}(t.\rho) \qquad \Omega \vdash \tau\ \mathsf{type}}{\Omega \vdash^{\mathsf{Pat}}_{\Phi} \mathtt{aptype}\{\tau\}(\epsilon) \mathbin{@} [\tau/t]\rho} \tag{5.16d}$$

$$\frac{\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mathbin{@} \mathtt{allmods}\{\sigma\}(X'.\rho) \qquad \Omega \vdash X : \sigma}{\Omega \vdash^{\mathsf{Pat}}_{\Phi} \mathtt{apmod}\{X\}(\epsilon) \mathbin{@} [X/X']\rho} \tag{5.16e}$$

peTSM Expression Expansion

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} \rho}{\langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega \rangle \vdash^{\mathsf{Exp}}_{\langle \mathcal{A}, \hat{a} \hookrightarrow \epsilon; \Psi; \mathcal{I} \rangle} \mathtt{bindref}[\hat{a}] \rightsquigarrow \epsilon \mathbin{@} \rho} \tag{5.17a}$$

$$\frac{\hat{\Omega}, \hat{t} \rightsquigarrow t :: \mathsf{Type} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \rho}{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \mathtt{uabstype}(\hat{t}.\hat{\epsilon}) \rightsquigarrow \mathtt{abstype}(t.\epsilon) \mathbin{@} \mathtt{alltypes}(t.\rho)} \tag{5.17b}$$

$$\frac{\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma\ \mathsf{sig} \qquad \hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \rho}{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \mathtt{uabsmod}\{\hat{\sigma}\}(\hat{X}.\hat{\epsilon}) \rightsquigarrow \mathtt{absmod}\{\sigma\}(X.\epsilon) \mathbin{@} \mathtt{allmods}\{\sigma\}(X.\rho)} \tag{5.17c}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \mathtt{alltypes}(t.\rho) \qquad \hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type}}{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \mathtt{uaptype}\{\hat{\tau}\}(\hat{\epsilon}) \rightsquigarrow \mathtt{aptype}\{\tau\}(\epsilon) \mathbin{@} [\tau/t]\rho} \tag{5.17d}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \mathtt{allmods}\{\sigma\}(X'.\rho) \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{X} \rightsquigarrow X \Leftarrow \sigma}{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \mathtt{uapmod}\{\hat{X}\}(\hat{\epsilon}) \rightsquigarrow \mathtt{apmod}\{X\}(\epsilon) \mathbin{@} [X/X']\rho} \tag{5.17e}$$

ppTSM expression expansion

$$\frac{\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mathbin{@} \rho}{\langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega \rangle \vdash^{\mathsf{Pat}}_{\langle \mathcal{A}, \hat{a} \hookrightarrow \epsilon; \Phi; \mathcal{I} \rangle} \mathtt{bindref}[\hat{a}] \rightsquigarrow \epsilon \mathbin{@} \rho} \tag{5.18a}$$

$$\frac{\hat{\Omega}, \hat{t} \rightsquigarrow t :: \mathsf{Type} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \rho}{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \mathtt{uabstype}(\hat{t}.\hat{\epsilon}) \rightsquigarrow \mathtt{abstype}(t.\epsilon) \mathbin{@} \mathtt{alltypes}(t.\rho)} \tag{5.18b}$$

$$\frac{\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma\ \mathsf{sig} \qquad \hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \rho}{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \mathtt{uabsmod}\{\hat{\sigma}\}(\hat{X}.\hat{\epsilon}) \rightsquigarrow \mathtt{absmod}\{\sigma\}(X.\epsilon) \mathbin{@} \mathtt{allmods}\{\sigma\}(X.\rho)} \tag{5.18c}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \mathtt{alltypes}(t.\rho) \qquad \hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type}}{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \mathtt{uaptype}\{\hat{\tau}\}(\hat{\epsilon}) \rightsquigarrow \mathtt{aptype}\{\tau\}(\epsilon) \mathbin{@} [\tau/t]\rho} \tag{5.18d}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \mathtt{allmods}\{\sigma\}(X'.\rho) \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{X} \rightsquigarrow X \Leftarrow \sigma}{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \mathtt{uapmod}\{\hat{X}\}(\hat{\epsilon}) \rightsquigarrow \mathtt{apmod}\{X\}(\epsilon) \mathbin{@} [X/X']\rho} \tag{5.18e}$$

peTSM expression normal forms

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi, a \hookrightarrow \mathtt{petsm}(\rho; e_{\mathrm{parse}})} \mathtt{defref}[a] \ \mathsf{normal}} \tag{5.19a}$$

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{abstype}(t.\epsilon) \ \mathsf{normal}} \tag{5.19b}$$

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{absmod}\{\sigma\}(X.\epsilon) \ \mathsf{normal}} \tag{5.19c}$$

$$\frac{\epsilon \neq \mathtt{abstype}(t.\epsilon') \qquad \Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \ \mathsf{normal}}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{aptype}\{\tau\}(\epsilon) \ \mathsf{normal}} \tag{5.19d}$$

$$\frac{\epsilon \neq \mathtt{absmod}\{\sigma\}(X'.\epsilon') \qquad \Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \ \mathsf{normal}}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{apmod}\{X\}(\epsilon) \ \mathsf{normal}} \tag{5.19e}$$

peTSM transitions

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto \epsilon'}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{aptype}\{\tau\}(\epsilon) \mapsto \mathtt{aptype}\{\tau\}(\epsilon')} \tag{5.20a}$$

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{aptype}\{\tau\}(\mathtt{abstype}(t.\epsilon)) \mapsto [\tau/t]\epsilon} \tag{5.20b}$$

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto \epsilon'}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{apmod}\{X\}(\epsilon) \mapsto \mathtt{apmod}\{X\}(\epsilon')} \tag{5.20c}$$

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{apmod}\{X\}(\mathtt{absmod}\{\sigma\}(X'.\epsilon)) \mapsto [X/X']\epsilon} \tag{5.20d}$$

peTSM reflexive, transitive transitions

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto^* \epsilon} \tag{5.21a}$$

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto \epsilon'}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto^* \epsilon'} \tag{5.21b}$$

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto^* \epsilon' \qquad \Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon' \mapsto^* \epsilon''}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto^* \epsilon''} \tag{5.21c}$$

peTSM normalization

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto^* \epsilon' \qquad \Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon' \ \mathsf{normal}}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \Downarrow \epsilon'} \tag{5.22}$$

$$\texttt{tsmdef}(\texttt{defref}[a]) = a \tag{5.23a}$$
$$\texttt{tsmdef}(\texttt{abstype}(t.\epsilon)) = \texttt{tsmdef}(\epsilon) \tag{5.23b}$$
$$\texttt{tsmdef}(\texttt{absmod}\{\sigma\}(X.\epsilon)) = \texttt{tsmdef}(\epsilon) \tag{5.23c}$$
$$\texttt{tsmdef}(\texttt{aptype}\{\tau\}(\epsilon)) = \texttt{tsmdef}(\epsilon) \tag{5.23d}$$
$$\texttt{tsmdef}(\texttt{apmod}\{X\}(\epsilon)) = \texttt{tsmdef}(\epsilon) \tag{5.23e}$$

5.2.6 Syntax of Candidate Expansions

Figure 5.13 defines the syntax of candidate expansion types (or *ce-types*), $\grave{\tau}$, candidate expansion expressions (or *ce-expressions*), $\grave{e}$, candidate expansion rules (or *ce-rules*), $\grave{r}$, and candidate expansion patterns (or *ce-patterns*), $\grave{p}$. Candidate expansion terms are identified up to α-equivalence in the usual manner.

Each inner core form, except for the variable pattern form, maps onto a candidate expansion form. In particular:

- Each type form maps onto a ce-type form according to the metafunction $\mathcal{P}(\tau)$, defined in Sec. 3.2.9.
- Each expanded expression form maps onto a ce-expression form according to the metafunction $\mathcal{P}(e)$, defined as follows:

$$\begin{aligned}
\mathcal{P}(x) &= x \\
\mathcal{P}(\texttt{lam}\{\tau\}(x.e)) &= \texttt{prlam}\{\mathcal{P}(\tau)\}(x.\mathcal{P}(e)) \\
\mathcal{P}(\texttt{ap}(e_1;e_2)) &= \texttt{prap}(\mathcal{P}(e_1);\mathcal{P}(e_2)) \\
\mathcal{P}(\texttt{tlam}(t.e)) &= \texttt{prtlam}(t.\mathcal{P}(e)) \\
\mathcal{P}(\texttt{tap}\{\tau\}(e)) &= \texttt{prtap}\{\mathcal{P}(\tau)\}(\mathcal{P}(e)) \\
\mathcal{P}(\texttt{fold}\{t.\tau\}(e)) &= \texttt{prasc}\{\texttt{prrec}(t.\mathcal{P}(\tau))\}(\texttt{prfold}(\mathcal{P}(e))) \\
\mathcal{P}(\texttt{unfold}(e)) &= \texttt{prunfold}(\mathcal{P}(e)) \\
\mathcal{P}(\texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L})) &= \texttt{prtpl}\{L\}(\{i \hookrightarrow \mathcal{P}(e_i)\}_{i\in L}) \\
\mathcal{P}(\texttt{inj}[L;\ell]\{\{i \hookrightarrow \tau_i\}_{i\in L}\}(e)) &= \texttt{prasc}\{\texttt{prsum}[L](\{i \hookrightarrow \mathcal{P}(\tau_i)\}_{i\in L})\}(\texttt{prin}[\ell](\mathcal{P}(e))) \\
\mathcal{P}(\texttt{match}[n]\{\tau\}(e;\{r_i\}_{1\le i\le n})) &= \texttt{prasc}\{\mathcal{P}(\tau)\}(\texttt{prmatch}[n](\mathcal{P}(e);\{\mathcal{P}(r_i)\}_{1\le i\le n}))
\end{aligned}$$

- The expanded rule form maps onto the ce-rule form according to the metafunction $\mathcal{P}(r)$, defined as follows:

$$\mathcal{P}(\texttt{rule}(p.e)) = \texttt{prrule}(p.\mathcal{P}(e))$$

- Each expanded pattern form, except for the variable pattern form, maps onto a ce-pattern form according to the metafunction $\mathcal{P}(p)$, defined in Sec. 4.2.8.

Sort		Operational Form	Stylized Form	Description
PCEExp	$\check{e}$ ::=	$\texttt{prexp}(\grave{e})$	$\grave{e}$	ce-expression
		$\texttt{prbindtype}(t.\check{e})$	$\Lambda t.\check{e}$	type binding
		$\texttt{prbindmod}(X.\check{e})$	$\Lambda X.\check{e}$	module binding

Figure 5.11: Abstract syntax of parameterized candidate expansion expressions in miniVerse$_\mathbf{P}$. Parameterized candidate expansion expressions are identified up to α-equivalence.

Sort		Operational Form	Stylized Form	Description
CEKind	$\grave{\kappa}$ ::=	$\texttt{prdarr}(\grave{\kappa}; u.\grave{\kappa})$	$(u :: \grave{\kappa}) \to \grave{\kappa}$	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
		$\texttt{prType}$	$\texttt{T}$	types
		$\texttt{prS}(\grave{\tau})$	$[{=}\grave{\tau}]$	singleton
		$\texttt{splicedk}[m; n]$	$\texttt{splicedk}\langle m, n\rangle$	spliced
CECon	$\grave{c}, \grave{\tau}$ ::=	u	u	constructor variable
		t	t	type variable
		$\texttt{prasc}\{\grave{\kappa}\}(\grave{c})$	$\grave{c} :: \grave{\kappa}$	ascription
		$\texttt{prabs}(u.\grave{c})$	$\lambda u.\grave{c}$	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
		$\texttt{prcon}(X)$	$X \cdot \texttt{c}$	constructor part
		$\texttt{splicedc}[m; n]$	$\texttt{splicedc}\langle m, n\rangle$	spliced

Figure 5.12: Syntax of candidate expansion kinds and constructors in miniVerse$_\mathbf{P}$. Candidate expansion kinds and constructors are identified up to α-equivalence.

Sort			Operational Form	Stylized Form	Description
CEExp	$\grave{e}$	::=	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
			$\texttt{pranalam}(x.\grave{e})$	$\lambda x.\grave{e}$	abstraction (unannotated)
			$\texttt{prlam}\{\grave{\tau}\}(x.\grave{e})$	$\lambda x{:}\grave{\tau}.\grave{e}$	abstraction (annotated)
			$\texttt{prap}(\grave{e}; \grave{e})$	$\grave{e}(\grave{e})$	application
			$\texttt{prclam}\{\grave{\kappa}\}(u.\grave{e})$	$\Lambda u{::}\grave{\kappa}.\grave{e}$	constructor abstraction
			$\texttt{prcap}\{\grave{c}\}(\grave{e})$	$\grave{e}[\grave{c}]$	constructor 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
			$\texttt{prin}[\ell](\grave{e})$	$\texttt{inj}[\ell](\grave{e})$	injection
			$\texttt{prmatch}[n](\grave{e}; \{\hat{r}_i\}_{1 \leq i \leq n})$	$\texttt{match}\ \grave{e}\ \{\hat{r}_i\}_{1 \leq i \leq n}$	match
			$\texttt{prval}(X)$	$X \cdot \texttt{v}$	value part
			$\texttt{splicede}[m; n]$	$\texttt{splicede}\langle m, n \rangle$	spliced
CERule	$\hat{r}$	::=	$\texttt{prrule}(p.\grave{e})$	$p \Rightarrow \grave{e}$	rule
CEPat	$\grave{p}$	::=	$\texttt{prwildp}$	$\_$	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
			$\texttt{splicedp}[m; n]$	$\texttt{splicedp}\langle m, n \rangle$	spliced

Figure 5.13: Abstract syntax of candidate expansion types, expressions, rules and patterns in miniVerse$_\mathbf{P}$. Candidate expansion terms are identified up to α-equivalence.

There are three other candidate expansion forms: a ce-type form for *references to spliced unexpanded types*, $\texttt{splicedt}[m;n]$, a ce-expression form for *references to spliced unexpanded expressions*, $\texttt{splicede}[m;n]$, and a ce-pattern form for *references to spliced unexpanded patterns*, $\texttt{splicedp}[m;n]$.

5.2.7 Candidate Expansion Validation

Judgement Form	**Description**
$\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \check{e} \looparrowright_{\epsilon} \grave{e} \mathbin{?} \rho \dashv \omega : \Omega_{\text{params}}$	$\check{e}$ has deparameterization $\grave{e} \mathbin{?} \rho$ when generated by ϵ, with substitution ω for variables tracked by Ω_{params}
$\Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi} \cdot \looparrowright_{\epsilon} \cdot \mathbin{?} \rho \dashv \omega : \Omega_{\text{params}}$	Any ce-pattern generated by ϵ has deparameterization $\cdot \mathbin{?} \rho$ with substitution ω for variables tracked by Ω_{params}

Candidate Expansion Expression Deparameterization

$$\frac{}{\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi, a \hookrightarrow \texttt{petsm}(\rho; e_{\text{parse}})} \texttt{prexp(}\grave{e}\texttt{)} \looparrowright_{\texttt{defref[}a\texttt{]}} \grave{e} \mathbin{?} \rho \dashv \emptyset : \emptyset} \tag{5.24a}$$

$$\frac{\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \check{e} \looparrowright_{\epsilon} \grave{e} \mathbin{?} \texttt{alltypes(}t.\rho\texttt{)} \dashv \omega : \Omega \qquad t \notin \text{dom}(\Omega_{\text{app}})}{\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \texttt{prbindtype(}t.\check{e}\texttt{)} \looparrowright_{\texttt{aptype\{}\tau\texttt{\}(}\epsilon\texttt{)}} \grave{e} \mathbin{?} \rho \dashv \omega, \tau/t : \Omega, t :: \texttt{Type}} \tag{5.24b}$$

$$\frac{\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \check{e} \looparrowright_{\epsilon} \grave{e} \mathbin{?} \texttt{allmods\{}\sigma\texttt{\}(}X.\rho\texttt{)} \dashv \omega : \Omega \qquad X \notin \text{dom}(\Omega_{\text{app}})}{\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \texttt{prbindmod(}X.\check{e}\texttt{)} \looparrowright_{\texttt{apmod\{}X'\texttt{\}(}\epsilon\texttt{)}} \grave{e} \mathbin{?} \rho \dashv \omega, X'/X : \Omega, X : \sigma} \tag{5.24c}$$

Candidate Expansion Pattern Deparameterization

$$\frac{}{\Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi, a \hookrightarrow \texttt{pptsm}(\rho; e_{\text{parse}})} \cdot \looparrowright_{\texttt{defref[}a\texttt{]}} \cdot \mathbin{?} \rho \dashv \emptyset : \emptyset} \tag{5.25a}$$

$$\frac{\Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi} \cdot \looparrowright_{\epsilon} \cdot \mathbin{?} \texttt{alltypes(}t.\rho\texttt{)} \dashv \omega : \Omega \qquad t \notin \text{dom}(\Omega_{\text{app}})}{\Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi} \cdot \looparrowright_{\texttt{aptype\{}\tau\texttt{\}(}\epsilon\texttt{)}} \cdot \mathbin{?} \rho \dashv \omega, \tau/t : \Omega, t :: \texttt{Type}} \tag{5.25b}$$

$$\frac{\Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi} \cdot \looparrowright_{\epsilon} \cdot \mathbin{?} \texttt{allmods\{}\sigma\texttt{\}(}X.\rho\texttt{)} \dashv \omega : \Omega \qquad X \notin \text{dom}(\Omega_{\text{app}})}{\Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi} \cdot \looparrowright_{\texttt{apmod\{}X'\texttt{\}(}\epsilon\texttt{)}} \cdot \mathbin{?} \rho \dashv \omega, X'/X : \Omega, X : \sigma} \tag{5.25c}$$

The *bidirectional candidate expansion validation judgements* validate ce-terms and simultaneously generate their final expansions.

Judgement Form	**Description**
$\Omega \vdash^{\mathbb{C}} \grave{\kappa} \rightsquigarrow \kappa \text{ kind}$	$\grave{\kappa}$ is well-formed and has expansion κ
$\Omega \vdash^{\mathbb{C}} \grave{c} \rightsquigarrow c \Rightarrow \kappa$	$\grave{c}$ has expansion c and synthesizes kind κ
$\Omega \vdash^{\mathbb{C}} \grave{c} \rightsquigarrow c \Leftarrow \kappa$	$\grave{c}$ has expansion c when analyzed against kind κ

Judgement Form	Description
$\Omega \vdash^{\mathbb{C}} \grave{\tau} \rightsquigarrow \tau$ type	$\grave{\tau}$ has expansion τ
$\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau$	$\grave{e}$ has expansion e and synthesizes type τ
$\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau$	$\grave{e}$ has expansion e when analyzed against type τ
$\Omega \vdash^{\mathbb{E}} \grave{r} \rightsquigarrow r \Rightarrow \tau \Mapsto \tau'$	$\grave{r}$ has expansion r and takes values of type τ to values of synthesized type τ'
$\Omega \vdash^{\mathbb{E}} \grave{r} \rightsquigarrow r \Leftarrow \tau \Mapsto \tau'$	$\grave{r}$ has expansion r and takes values of type τ to values of type τ' when τ' is provided for analysis
$\Omega_{\text{param}} \vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : \tau \dashv\!\vert\vert^{\hat{\Omega}}$	$\grave{p}$ expands to p and matches values of type τ generating assumptions $\hat{\Gamma}$

Expression splicing scenes, $\mathbb{E}$, are of the form $\hat{\Omega}; \hat{\Psi}; \hat{\Phi}; b$, *constructor splicing scenes*, $\mathbb{C}$, are of the form $\hat{\Omega}; b$, and *pattern splicing scenes*, $\mathbb{P}$, are of the form $\hat{\Omega}; \hat{\Phi}; b$. Their purpose is to "remember", during candidate expansion validation, the contexts, TSM environments and literal bodies from the TSM application site (cf. Rules (**??**) and (B.8f)), because these are necessary to validate references to spliced terms. We write $\mathsf{cs}(\mathbb{E})$ for the constructor splicing scene constructed by dropping the TSM contexts from $\mathbb{E}$:

$$\mathsf{cs}(\hat{\Omega}; \hat{\Psi}; \hat{\Phi}; b) = \hat{\Omega}; b$$

Candidate Expansion Kind and Constructor Validation

ce-kind validation

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{\kappa}_1 \rightsquigarrow \kappa_1 \text{ kind} \qquad \Omega, u :: \kappa_1 \vdash^{\mathbb{C}} \grave{\kappa}_2 \rightsquigarrow \kappa_2 \text{ kind}}{\Omega \vdash^{\mathbb{C}} \texttt{prdarr}(\grave{\kappa}_1; u.\grave{\kappa}_2) \rightsquigarrow \texttt{darr}(\kappa_1; u.\kappa_2) \text{ kind}} \tag{5.26a}$$

$$\frac{}{\Omega \vdash^{\mathbb{C}} \texttt{prunit} \rightsquigarrow \texttt{unit} \text{ kind}} \tag{5.26b}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{\kappa}_1 \rightsquigarrow \kappa_1 \text{ kind} \qquad \Omega, u :: \kappa_1 \vdash^{\mathbb{C}} \grave{\kappa}_2 \rightsquigarrow \kappa_2 \text{ kind}}{\Omega \vdash^{\mathbb{C}} \texttt{prdprod}(\grave{\kappa}_1; u.\grave{\kappa}_2) \rightsquigarrow \texttt{dprod}(\kappa_1; u.\kappa_2) \text{ kind}} \tag{5.26c}$$

$$\frac{}{\Omega \vdash^{\mathbb{C}} \texttt{prType} \rightsquigarrow \texttt{Type} \text{ kind}} \tag{5.26d}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{\tau} \rightsquigarrow \tau \Leftarrow \texttt{Type}}{\Omega \vdash^{\mathbb{C}} \texttt{prS}(\grave{\tau}) \rightsquigarrow \texttt{S}(\tau) \text{ kind}} \tag{5.26e}$$

$$\frac{\begin{array}{c}\mathsf{parseUKind}(\mathsf{subseq}(b; m; n)) = \hat{\kappa} \qquad \hat{\Omega} \vdash \hat{\kappa} \rightsquigarrow \kappa \text{ kind} \\ \hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega_{\text{app}} \rangle \qquad \mathrm{dom}(\Omega) \cap \mathrm{dom}(\Omega_{\text{app}}) = \emptyset\end{array}}{\Omega \vdash^{\hat{\Omega}; b} \texttt{splicedk}[m; n] \rightsquigarrow \kappa \text{ kind}} \tag{5.26f}$$

Synthetic ce-constructor validation

$$\frac{}{\Omega, u :: \kappa \vdash^{\mathbb{C}} u \rightsquigarrow u \Rightarrow \kappa} \tag{5.27a}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{\kappa} \rightsquigarrow \kappa \;\mathsf{kind} \qquad \Omega \vdash^{\mathbb{C}} \grave{c} \rightsquigarrow c \Leftarrow \kappa}{\Omega \vdash^{\mathbb{C}} \mathtt{prasc\{\grave{\kappa}\}(\grave{c})} \rightsquigarrow c \Rightarrow \kappa} \tag{5.27b}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{c}_1 \rightsquigarrow c_1 \Rightarrow \mathtt{darr}(\kappa_2; u.\kappa) \qquad \Omega \vdash^{\mathbb{C}} \grave{c}_2 \rightsquigarrow c_2 \Rightarrow \kappa_2}{\Omega \vdash^{\mathbb{C}} \mathtt{prapp}(\grave{c}_1; \grave{c}_2) \rightsquigarrow \mathtt{app}(c_1; c_2) \Rightarrow [c_1/u]\kappa} \tag{5.27c}$$

$$\frac{}{\Omega \vdash^{\mathbb{C}} \mathtt{prtriv} \rightsquigarrow \mathtt{triv} \Rightarrow \mathtt{unit}} \tag{5.27d}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{c} \rightsquigarrow c \Rightarrow \mathtt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash^{\mathbb{C}} \mathtt{prprl}(\grave{c}) \rightsquigarrow \mathtt{prl}(c) \Rightarrow \kappa_1} \tag{5.27e}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{c} \rightsquigarrow c \Rightarrow \mathtt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash^{\mathbb{C}} \mathtt{prprr}(\grave{c}) \rightsquigarrow \mathtt{prr}(c) \Rightarrow [\mathtt{prl}(c)/u]\kappa_2} \tag{5.27f}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{\tau}_1 \rightsquigarrow \tau_1 \Leftarrow \mathtt{Type} \qquad \Omega \vdash^{\mathbb{C}} \grave{\tau}_2 \rightsquigarrow \tau_2 \Leftarrow \mathtt{Type}}{\Omega \vdash^{\mathbb{C}} \mathtt{prparr}(\grave{\tau}_1; \grave{\tau}_2) \rightsquigarrow \mathtt{parr}(\tau_1; \tau_2) \Rightarrow \mathtt{Type}} \tag{5.27g}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{\kappa} \rightsquigarrow \kappa \;\mathsf{kind} \qquad \Omega, u :: \kappa \vdash^{\mathbb{C}} \grave{\tau} \rightsquigarrow \tau \Leftarrow \mathtt{Type}}{\Omega \vdash^{\mathbb{C}} \mathtt{prall\{\grave{\kappa}\}(u.\grave{\tau})} \rightsquigarrow \mathtt{all\{\kappa\}(u.\tau)} \Rightarrow \mathtt{Type}} \tag{5.27h}$$

$$\frac{\Omega, t :: \mathtt{Type} \vdash^{\mathbb{C}} \grave{\tau} \rightsquigarrow \tau \Leftarrow \mathtt{Type}}{\Omega \vdash^{\mathbb{C}} \mathtt{prrec}(t.\grave{\tau}) \rightsquigarrow \mathtt{rec}(t.\tau) \Rightarrow \mathtt{Type}} \tag{5.27i}$$

$$\frac{\{\Omega \vdash^{\mathbb{C}} \grave{\tau}_i \rightsquigarrow \tau_i \Leftarrow \mathtt{Type}\}_{1 \leq i \leq n}}{\Omega \vdash^{\mathbb{C}} \mathtt{uprod}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}) \rightsquigarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \Rightarrow \mathtt{Type}} \tag{5.27j}$$

$$\frac{\{\Omega \vdash^{\mathbb{C}} \grave{\tau}_i \rightsquigarrow \tau_i \Leftarrow \mathtt{Type}\}_{1 \leq i \leq n}}{\Omega \vdash^{\mathbb{C}} \mathtt{prsum}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}) \rightsquigarrow \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \Rightarrow \mathtt{Type}} \tag{5.27k}$$

$$\frac{}{\Omega, X : \mathtt{sig\{\kappa\}(u.\tau)} \vdash^{\mathbb{C}} \mathtt{prcon}(X) \rightsquigarrow \mathtt{con}(X) \Rightarrow \kappa} \tag{5.27l}$$

$$\frac{\mathsf{parseUCon}(\mathsf{subseq}(b; m; n)) = \hat{c} \qquad \hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Rightarrow \kappa \qquad \hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega_{\mathrm{app}} \rangle \qquad \mathrm{dom}(\Omega) \cap \mathrm{dom}(\Omega_{\mathrm{app}}) = \emptyset}{\Omega \vdash^{\hat{\Omega}; b} \mathtt{splicedc}[m; n] \rightsquigarrow c \Rightarrow \kappa} \tag{5.27m}$$

Analytic constructor expansion

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{c} \rightsquigarrow c \Rightarrow \kappa_1 \qquad \Omega \vdash \kappa_1 <:: \kappa_2}{\Omega \vdash^{\mathbb{C}} \grave{c} \rightsquigarrow c \Leftarrow \kappa_2} \tag{5.28a}$$

$$\frac{\hat{\Omega} \vdash \grave{c} \rightsquigarrow c \Leftarrow \mathtt{Type}}{\hat{\Omega} \vdash \grave{c} \rightsquigarrow c \Leftarrow \mathtt{S}(c)} \tag{5.28b}$$

$$\frac{\Omega, u :: \kappa_1 \vdash^{\mathbb{C}} \grave{c}_2 \rightsquigarrow c_2 \Leftarrow \kappa_2}{\Omega \vdash^{\mathbb{C}} \mathtt{prabs}(u.\grave{c}_2) \rightsquigarrow \mathtt{abs}(u.c_2) \Leftarrow \mathtt{darr}(\kappa_1; u.\kappa_2)} \tag{5.28c}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{c}_1 \rightsquigarrow c_1 \Leftarrow \kappa_1 \qquad \Omega \vdash^{\mathbb{C}} \grave{c}_2 \rightsquigarrow c_2 \Leftarrow [c_1/u]\kappa_2}{\Omega \vdash^{\mathbb{C}} \mathtt{prpair}(\grave{c}_1; \grave{c}_2) \rightsquigarrow \mathtt{pair}(c_1; c_2) \Leftarrow \mathtt{dprod}(\kappa_1; u.\kappa_2)} \tag{5.28d}$$

$$\frac{\mathsf{parseUCon}(\mathsf{subseq}(b; m; n)) = \hat{c} \qquad \hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Leftarrow \kappa \qquad \hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega_{\mathrm{app}} \rangle \qquad \mathrm{dom}(\Omega) \cap \mathrm{dom}(\Omega_{\mathrm{app}}) = \emptyset}{\Omega \vdash^{\hat{\Omega}; b} \mathtt{splicedc}[m; n] \rightsquigarrow c \Leftarrow \kappa} \tag{5.28e}$$

Bidirectional Candidate Expansion Expression Validation

Like the bidirectionally typed expression expansion judgements, the bidirectional ce-expression validation judgements distinguish type synthesis from type analysis. The *synthetic ce-expression validation judgement*, $\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau$, and the *analytic ce-expression validation judgement*, $\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau$, are defined mutually inductively with Rules (7.1) and Rules (7.3) by Rules (7.8) and Rules (7.9), respectively, as follows.

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{\tau} \rightsquigarrow \tau \Leftarrow \mathsf{Type}}{\Omega \vdash^{\mathbb{C}} \grave{\tau} \rightsquigarrow \tau\ \mathsf{type}} \tag{5.29}$$

Type Synthesis Synthetic ce-expression validation is governed by the following rules.

$$\frac{}{\Omega, x : \tau \vdash^{\mathbb{E}} x \rightsquigarrow x \Rightarrow \tau} \tag{5.30a}$$

$$\frac{\Omega \vdash^{\mathsf{cs}(\mathbb{E})} \grave{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau}{\Omega \vdash^{\mathbb{E}} \mathtt{prasc}\{\grave{\tau}\}(\grave{e}) \rightsquigarrow e \Rightarrow \tau} \tag{5.30b}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \Omega, x : \tau \vdash^{\mathbb{E}} \grave{e}' \rightsquigarrow e' \Rightarrow \tau'}{\Omega \vdash^{\mathbb{E}} \mathtt{prletval}(\grave{e}; x.\grave{e}') \rightsquigarrow \mathtt{ap}(\mathtt{lam}\{\tau\}(x.e'); e) \Rightarrow \tau'} \tag{5.30c}$$

$$\frac{\Omega \vdash^{\mathsf{cs}(\mathbb{E})} \grave{\tau}_1 \rightsquigarrow \tau_1\ \mathsf{type} \qquad \Omega, x : \tau_1 \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau_2}{\Omega \vdash^{\mathbb{E}} \mathtt{prlam}\{\grave{\tau}_1\}(x.\grave{e}) \rightsquigarrow \mathtt{lam}\{\tau_1\}(x.e) \Rightarrow \mathtt{parr}(\tau_1; \tau_2)} \tag{5.30d}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e}_1 \rightsquigarrow e_1 \Rightarrow \mathtt{parr}(\tau_2; \tau) \qquad \Omega \vdash^{\mathbb{E}} \grave{e}_2 \rightsquigarrow e_2 \Leftarrow \tau_2}{\Omega \vdash^{\mathbb{E}} \mathtt{prap}(\grave{e}_1; \grave{e}_2) \rightsquigarrow \mathtt{ap}(e_1; e_2) \Rightarrow \tau} \tag{5.30e}$$

$$\frac{\Omega \vdash^{\mathsf{cs}(\mathbb{E})} \grave{\kappa} \rightsquigarrow \kappa\ \mathsf{kind} \qquad \Omega, u :: \kappa \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prclam}\{\grave{\kappa}\}(u.\grave{e}) \rightsquigarrow \mathtt{clam}\{\kappa\}(u.e) \Rightarrow \mathtt{all}\{\kappa\}(u.\tau)} \tag{5.30f}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \mathtt{all}\{\kappa\}(u.\tau) \qquad \Omega \vdash^{\mathsf{cs}(\mathbb{E})} \grave{c} \rightsquigarrow c \Leftarrow \kappa}{\Omega \vdash^{\mathbb{E}} \mathtt{prcap}\{\grave{c}\}(\grave{e}) \rightsquigarrow \mathtt{cap}\{c\}(e) \Rightarrow [c/u]\tau} \tag{5.30g}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \mathtt{rec}(t.\tau)}{\Omega \vdash^{\mathbb{E}} \mathtt{prunfold}(\grave{e}) \rightsquigarrow \mathtt{unfold}(e) \Rightarrow [\mathtt{rec}(t.\tau)/t]\tau} \tag{5.30h}$$

$$\frac{\grave{e} = \mathtt{prtpl}\{L\}(\{i \hookrightarrow \grave{e}_i\}_{i \in L}) \qquad e = \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) \qquad \{\Omega \vdash^{\mathbb{E}} \grave{e}_i \rightsquigarrow e_i \Rightarrow \tau_i\}_{i \in L}}{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{5.30i}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \mathtt{prod}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)}{\Omega \vdash^{\mathbb{E}} \mathtt{prprj}[\ell](\grave{e}) \rightsquigarrow \mathtt{prj}[\ell](e) \Rightarrow \tau} \tag{5.30j}$$

$$\frac{n > 0 \qquad \Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \{\Omega \vdash^{\mathbb{E}} \grave{r}_i \rightsquigarrow r_i \Rightarrow \tau \Mapsto \tau'\}_{1 \leq i \leq n}}{\Omega \vdash^{\mathbb{E}} \mathtt{prmatch}[n](\grave{e}; \{\grave{r}_i\}_{1 \leq i \leq n}) \rightsquigarrow \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \leq i \leq n}) \Rightarrow \tau'} \tag{5.30k}$$

$$\frac{}{\Omega, X : \mathtt{sig}\{\kappa\}(u.\tau) \vdash^{\mathbb{E}} \mathtt{prval}(X) \rightsquigarrow \mathtt{val}(X) \Rightarrow [\mathtt{con}(X)/u]\tau} \tag{5.30l}$$

$$\frac{\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e} \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega_{\mathrm{app}} \rangle \qquad \mathrm{dom}(\Omega) \cap \mathrm{dom}(\Omega_{\mathrm{app}}) = \emptyset}{\Omega \vdash^{\hat{\Omega};\hat{\Psi};\hat{\Phi};b} \mathtt{splicede}[m; n] \rightsquigarrow e \Rightarrow \tau} \tag{5.30m}$$

Rules (7.8a) through (7.8k) are analagous to Rules (7.1a) through (7.1k). Rule (7.8l) governs references to spliced unexpanded expressions in synthetic position, and can be understood as described in Sec. 3.2.10.

Type Analysis Analytic ce-expression validation is governed by the following rules.

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \Omega \vdash \tau <: \tau'}{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau'} \tag{5.31a}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \Omega, x : \tau \vdash^{\mathbb{E}} \grave{e}' \rightsquigarrow e' \Leftarrow \tau'}{\Omega \vdash^{\mathbb{E}} \mathtt{prletval}(\grave{e}; x.\grave{e}') \rightsquigarrow \mathtt{ap}(\mathtt{lam}\{\tau\}(x.e'); e) \Leftarrow \tau'} \tag{5.31b}$$

$$\frac{\Gamma, x : \tau_1 \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau_2}{\Omega \vdash^{\mathbb{E}} \mathtt{pranalam}(x.\hat{e}) \rightsquigarrow \mathtt{lam}\{\tau_1\}(x.e) \Leftarrow \mathtt{parr}(\tau_1; \tau_2)} \tag{5.31c}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{\kappa} \rightsquigarrow \kappa \ \mathsf{kind} \qquad \Omega, u :: \kappa \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau}{\Omega \vdash^{\mathbb{E}} \mathtt{prclam}\{\grave{\kappa}\}(u.\grave{e}) \rightsquigarrow \mathtt{clam}\{\kappa\}(u.e) \Leftarrow \mathtt{all}\{\kappa\}(u.\tau)} \tag{5.31d}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow [\mathtt{rec}(t.\tau)/t]\tau}{\Omega \vdash^{\mathbb{E}} \mathtt{prfold}(\grave{e}) \rightsquigarrow \mathtt{fold}\{t.\tau\}(e) \Leftarrow \mathtt{rec}(t.\tau)} \tag{5.31e}$$

$$\frac{\grave{e} = \mathtt{prtpl}\{L\}(\{i \hookrightarrow \grave{e}_i\}_{i \in L}) \qquad e = \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) \qquad \{\Omega \vdash^{\mathbb{E}} \grave{e}_i \rightsquigarrow e_i \Leftarrow \tau_i\}_{i \in L}}{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{5.31f}$$

$$\frac{\grave{e} = \mathtt{prin}[\ell](\grave{e}') \qquad e = \mathtt{inj}[L,\ell;\ell]\{\{i \hookrightarrow \tau_i\}_{i\in L};\ell \hookrightarrow \tau\}(e') \qquad \Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e}' \rightsquigarrow e' \Leftarrow \tau}{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \mathtt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i\in L};\ell \hookrightarrow \tau)} \tag{5.31g}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \{\Omega \vdash^{\mathbb{E}} \grave{r}_i \rightsquigarrow r_i \Leftarrow \tau \Mapsto \tau'\}_{1\le i\le n}}{\Omega \vdash^{\mathbb{E}} \mathtt{prmatch}[n](\grave{e};\{\grave{r}_i\}_{1\le i\le n}) \rightsquigarrow \mathtt{match}[n]\{\tau'\}(e;\{r_i\}_{1\le i\le n}) \Leftarrow \tau'} \tag{5.31h}$$

$$\frac{\mathsf{parseUExp}(\mathsf{subseq}(b;m;n)) = \hat{e} \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau \qquad \hat{\Omega} = \langle \mathcal{D};\mathcal{G};\mathcal{M};\Omega_{\mathrm{app}}\rangle \qquad \mathrm{dom}(\Omega) \cap \mathrm{dom}(\Omega_{\mathrm{app}}) = \emptyset}{\Omega \vdash^{\hat{\Omega};\hat{\Psi};\hat{\Phi};b} \mathtt{splicede}[m;n] \rightsquigarrow e \Leftarrow \tau} \tag{5.31i}$$

Rules (7.9a) through (7.9h) are analagous to Rules (7.3a) through (7.3h). Rule (7.9i) governs references to spliced unexpanded expressions in analytic position.

Bidirectional Candidate Expansion Rule Validation

The *synthetic ce-rule validation judgement* is defined mutually inductively with Rules (7.1) by the following rule.

$$\frac{\Omega \vdash p : \tau \dashv\!\!\mid \Omega' \qquad \Omega \cup \Omega' \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau'}{\Omega \vdash^{\mathbb{E}} \mathtt{prrule}(p.\grave{e}) \rightsquigarrow \mathtt{rule}(p.e) \Rightarrow \tau \Mapsto \tau'} \tag{5.32}$$

The *analytic ce-rule validation judgement* is defined mutually inductively with Rules (7.3) by the following rule.

$$\frac{\Delta \vdash p : \tau \dashv\!\!\mid \Omega' \qquad \Omega \cup \Omega' \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau'}{\Omega \vdash^{\mathbb{E}} \mathtt{prrule}(p.\grave{e}) \rightsquigarrow \mathtt{rule}(p.e) \Leftarrow \tau \Mapsto \tau'} \tag{5.33}$$

Candidate Expansion Pattern Validation

The *ce-pattern validation judgement* is inductively defined by the following rules, which are written identically to Rules (B.12).

$$\frac{}{\Omega_{\mathrm{param}} \vdash^{\mathbb{P}} \mathtt{prwildp} \rightsquigarrow \mathtt{wildp} : \tau \dashv\!\!\mid^{\langle\emptyset;\emptyset;\emptyset;\emptyset\rangle}} \tag{5.34a}$$

$$\frac{\Omega_{\mathrm{param}} \vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : [\mathtt{rec}(t.\tau)/t]\tau \dashv\!\!\mid^{\hat{\Omega}}}{\Omega_{\mathrm{param}} \vdash^{\mathbb{P}} \mathtt{prfoldp}(\grave{p}) \rightsquigarrow \mathtt{foldp}(p) : \mathtt{rec}(t.\tau) \dashv\!\!\mid^{\hat{\Omega}}} \tag{5.34b}$$

$$\frac{\grave{p} = \mathtt{prtplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i\in L}) \qquad p = \mathtt{tplp}[L](\{i \hookrightarrow p_i\}_{i\in L}) \qquad \{\Omega_{\mathrm{param}} \vdash^{\mathbb{P}} \grave{p}_i \rightsquigarrow p_i : \tau_i \dashv\!\!\mid^{\hat{\Gamma}_i}\}_{i\in L}}{\Omega_{\mathrm{param}} \vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L}) \dashv\!\!\mid^{\cup_{i\in L}\hat{\Omega}_i}} \tag{5.34c}$$

$$\frac{\Omega_{\text{param}} \vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : \tau \dashv\!\!\mid^{\hat{\Omega}}}{\Omega_{\text{param}} \vdash^{\mathbb{P}} \texttt{prinjp}[\ell]\texttt{(}\grave{p}\texttt{)} \rightsquigarrow \texttt{injp}[\ell]\texttt{(}p\texttt{)} : \texttt{sum}[L,\ell]\texttt{(}\{i \hookrightarrow \tau_i\}_{i\in L}; \ell \hookrightarrow \tau\texttt{)} \dashv\!\!\mid^{\hat{\Omega}}} \tag{5.34d}$$

$$\frac{\mathsf{parseUPat}(\mathsf{subseq}(b;m;n)) = \hat{p} \qquad \hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\!\mid \hat{\Omega}'}{\Omega_{\text{param}} \vdash^{\hat{\Omega};\hat{\Phi};b} \texttt{splicedp}[m;n] \rightsquigarrow p : \tau \dashv\!\!\mid^{\hat{\Omega}'}} \tag{5.34e}$$

5.2.8 Metatheory

Chapter 6

Static Evaluation and State

In the previous sections, we have assumed that the parse functions in a TSM definition are closed expanded expressions. This is unrealistic. In this section, we discuss the semantics of the static phase of evaluation. We also add support for stateful programming with reference cells, so that we can discuss how these interact with static evaluation.

6.1 TSMs For Defining TSMs

Static functions can also make use of TSMs. In this section, we will show how quasiquotation syntax and grammar-based parser generators can be expressed using TSMs. These TSMs are quite useful for writing other TSMs.

6.1.1 Quasiquotation

TSMs must generate values of type `CEExp`. Doing so explicitly can have high syntactic cost. To decrease the syntactic cost of constructing values of this type, the prelude includes a TSM that provides quasiquotation syntax (cf. Sec. **??**):

```
syntax $qqexp at CEExp {
  static fn(body : Body) : ParseResult => (* expression parser here *)
}

syntax $qqtype at CETyp {
  static fn(body : Body) : ParseResult => (* type parser here *)
}
```

For example, the following expression:

```
let gx = $qqexp `g(x)`
```

is more concise than its expansion:

```
let gx = App(Var 'g', Var 'x')
```

The full concrete syntax of the language can be used. Anti-quotation, i.e. splicing in an expression of type `MarkedExp`, is indicated by the prefix `%`:

```
let fgx = $qqexp `f(%gx)`
```

The expansion of this expression is:

```
let fgx = App(Var 'f', gx)
```

6.1.2 Parser Generators

TODO: grammars, compile function, TSM for grammar, example of IP address

6.2 Static Language

We have assumed throughout this work that parse functions are fully self-contained, i.e. they are closed. This simplifies our exposition and metatheory, but it is not a realistic constraint – in practice, one would want to be able to share helper code between parse functions. To allow this, VerseML allows programmers to introduce *static blocks*, which introduce bindings available only in other static blocks and static functions. For example, the following static block defines a helper function for use in the subsequent parse function.

```
1 static
2   val parseInt : Body -> Exp option = (* ... *)
3 end
4 syntax $rx at Rx {
5   static fn(body : Body) : ParseResultExp =>
6     (* ... parseInt is available here ... *)
7 }
8 val x = parseInt("34") (* error: parseInt is not bound here *)
```

Part III

TSM Implicits

Chapter 7

Unparameterized TSM Implicits

Using TSMs, a library provider can control the expansion of generalized literal forms, and thereby control the syntactic cost of common idioms. However, library clients must explicitly prefix each such form with a TSM name. In situations where the client is repeatedly using a TSM throughout a codebase, this can be inelegant. 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.

7.1 TSM Implicits By Example

We begin in this section by introducing TSM implicits by example in VerseML. In Sec. 7.2, we formalize unparameterized TSM implicits with a reduced calculus, $\text{miniVerse}^{\mathbf{B}}_{\mathbf{U}}$. We will also return to the topic of TSM implicits after introducing parameterized TSMs in Chapter 5.

7.1.1 Designation

In the example in Figure 7.1, Lines 1 through 3 designate the expression TSM named `$rx`, defined in Section 3.1.2, and the pattern TSM named `$rx`, defined in Sec. 4.1.2, both at type `Rx`. These designations influence typed expansion of Lines 5 through 9.

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
```

The type annotation on a designation is technically redundant – the definition of the designated TSM determines the designated type. It is included in our examples for readability, but can be omitted if desired.

```
1  implicit syntax
2    $rx at Rx for expressions
3    $rx at Rx for patterns
4  in
5    fun is_ssn(s : string) => rx_match /\d\d\d-\d\d-\d\d\d\d/ s
6    fun name_from_example_rx(r : Rx) : string option =>
7      match r with
8        /@name: %_/ => Some name
9      | _ => None
10 end
```

Figure 7.1: An example of TSM implicits in VerseML

7.1.2 Usage

On Line 5 of Figure 7.1, we apply a function `rx_match` (not shown), which has type `Rx -> string -> MatchResult`, to an expression of unadorned literal form. During typed expansion, the expression TSM `$rx` is applied implicitly to this form to determine the expression's expansion, because `$rx` is the designated TSM at the argument type `Rx`.

Similarly, a pattern of unadorned literal form appears on Line 8. Because it appears in a syntactic position where it must match values of type `Rx`, the pattern TSM `$rx` is implicitly applied to determine its expansion.

7.1.3 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 necessarily have can be determined based on the surrounding context, without examining 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 "further up" in the expression:

```
val ssn = /\d\d\d-\d\d-\d\d\d\d/ : Rx
val ssn : Rx = /\d\d\d-\d\d-\d\d\d\d/
fun ssn() : Rx => /\d\d\d-\d\d-\d\d\d\d/
```

If the type that e' must be assigned cannot be determined from 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 appearing in a binding or function definition without a type ascription, appears in synthetic position.

Expressions of unadorned literal form can only appear in analytic position, because their type must be known to be able to determine the designated TSM that will control their expansion. For example, typed expansion of the following expression will fail because subexpressions of unadorned literal form appear in synthetic position:

```
let
  val ssn = /\d\d\d-\d\d-\d\d\d\d/ (* INVALID *)
  fun ssn() => /\d\d\d-\d\d-\d\d\d\d/ (* INVALID *)
in
  (* ... *)
end
```

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 without examining the pattern itself.

7.2 $\mathsf{miniVerse}^{\mathbf{B}}_{\mathbf{U}}$

To formalize TSM implicits, we will now develop a reduced calculus called $\mathsf{miniVerse}^{\mathbf{B}}_{\mathbf{U}}$, or "Bidirectional $\mathsf{miniVerse}_{\mathbf{S}}$" (so named because it explicitly distinguishes type analysis from type synthesis during typed expansion, as explained below).

7.2.1 Inner Core

The inner core of $\mathsf{miniVerse}^{\mathbf{B}}_{\mathbf{U}}$ is the same as the inner core of $\mathsf{miniVerse}_{\mathbf{S}}$, as described in Sections 4.2.1 through 4.2.3. It consists of types, τ, expanded expressions, e, expanded rules, r, and expanded patterns, p.

7.2.2 Syntax of the Outer Surface

A $\mathsf{miniVerse}^{\mathbf{B}}_{\mathbf{U}}$ program ultimately evaluates as an expanded expression. However, the programmer does not write the expanded expression directly. Instead, the programmer writes a textual sequence, b, consisting of characters in some suitable alphabet (e.g. in practice, `ASCII` or `Unicode`), which is parsed by some partial metafunction $\mathsf{parseUExp}(b)$ to produce an *unexpanded expression*, $\hat{e}$. Unexpanded expressions can contain *unexpanded types*, $\hat{\tau}$, *unexpanded rules*, $\hat{r}$, and *unexpanded patterns*, $\hat{p}$, so we also need partial metafunctions $\mathsf{parseUTyp}(b)$, $\mathsf{parseURule}(b)$ and $\mathsf{parseUPat}(b)$. The abstract syntax of unexpanded types, expressions, rules and patterns, which form the *outer surface* of $\mathsf{miniVerse}^{\mathbf{B}}_{\mathbf{U}}$, is defined in Figure 4.3. The full definition of the textual syntax of $\mathsf{miniVerse}^{\mathbf{B}}_{\mathbf{U}}$ is not important for our purposes, so we simply give the following condition, which states that there is some way to textually represent every unexpanded type, expression, rule and pattern.

Condition 7.1 (Textual Representability)**.** *All of the following must hold:*

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* $\hat{r}$*, there exists* b *such that* $\mathsf{parseURule}(b) = \hat{r}$.
4. *For each* $\hat{p}$*, there exists* b *such that* $\mathsf{parseUPat}(b) = \hat{p}$.

As in $\mathsf{miniVerse}_{\mathbf{S}}$, unexpanded types and expressions bind *type sigils*, $\hat{t}$, *expression sigils*, $\hat{x}$, and *TSM names*, $\hat{a}$. Sigils are given meaning by expansion to variables during typed

Sort			Operational Form	Stylized Form	Description
UTyp	$\hat{\tau}$	::=	$\hat{t}$	$\hat{t}$	sigil
			$\texttt{uparr}(\hat{\tau}; \hat{\tau})$	$\hat{\tau} \rightharpoonup \hat{\tau}$	partial function
			$\texttt{uall}(\hat{t}.\hat{\tau})$	$\forall \hat{t}.\hat{\tau}$	polymorphic
			$\texttt{urec}(\hat{t}.\hat{\tau})$	$\mu \hat{t}.\hat{\tau}$	recursive
			$\texttt{uprod}[L](\{i \hookrightarrow \hat{\tau}_i\}_{i \in L})$	$\langle \{i \hookrightarrow \hat{\tau}_i\}_{i \in L} \rangle$	labeled product
			$\texttt{usum}[L](\{i \hookrightarrow \hat{\tau}_i\}_{i \in L})$	$[\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}]$	labeled sum
UExp	$\hat{e}$	::=	$\hat{x}$	$\hat{x}$	sigil
			$\texttt{uasc}\{\hat{\tau}\}(\hat{e})$	$\hat{e} : \hat{\tau}$	ascription
			$\texttt{uletval}(\hat{e}; \hat{x}.\hat{e})$	$\texttt{let val}\ \hat{x} = \hat{e}\ \texttt{in}\ \hat{e}$	value binding
			$\texttt{uanalam}(\hat{x}.\hat{e})$	$\lambda \hat{x}.\hat{e}$	abstraction (unannotated)
			$\texttt{ulam}\{\hat{\tau}\}(\hat{x}.\hat{e})$	$\lambda \hat{x}{:}\hat{\tau}.\hat{e}$	abstraction (annotated)
			$\texttt{uap}(\hat{e}; \hat{e})$	$\hat{e}(\hat{e})$	application
			$\texttt{utlam}(\hat{t}.\hat{e})$	$\Lambda \hat{t}.\hat{e}$	type abstraction
			$\texttt{utap}\{\hat{\tau}\}(\hat{e})$	$\hat{e}[\hat{\tau}]$	type application
			$\texttt{ufold}(\hat{e})$	$\texttt{fold}(\hat{e})$	fold
			$\texttt{uunfold}(\hat{e})$	$\texttt{unfold}(\hat{e})$	unfold
			$\texttt{utpl}\{L\}(\{i \hookrightarrow \hat{e}_i\}_{i \in L})$	$\langle \{i \hookrightarrow \hat{e}_i\}_{i \in L} \rangle$	labeled tuple
			$\texttt{uprj}[\ell](\hat{e})$	$\hat{e} \cdot \ell$	projection
			$\texttt{uin}[\ell](\hat{e})$	$\texttt{inj}[\ell](\hat{e})$	injection
			$\texttt{umatch}[n](\hat{e}; \{\hat{r}_i\}_{1 \leq i \leq n})$	$\texttt{match}\ \hat{e}\ \{\hat{r}_i\}_{1 \leq i \leq n}$	match
			$\texttt{usyntaxue}\{e\}\{\hat{\tau}\}(\hat{a}.\hat{e})$	$\texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for expressions}\ \{e\}\ \texttt{in}\ \hat{e}$	ueTSM definition
			$\texttt{uimplicite}[\hat{a}](\hat{e})$	$\texttt{implicit syntax}\ \hat{a}\ \texttt{for expressions in}\ \hat{e}$	ueTSM designation
			$\texttt{uapuetsm}[b][\hat{a}]$	$\hat{a}\ /b/$	ueTSM application
			$\texttt{uelit}[b]$	$/b/$	ueTSM unadorned literal
			$\texttt{usyntaxup}\{e\}\{\hat{\tau}\}(\hat{a}.\hat{e})$	$\texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for patterns}\ \{e\}\ \texttt{in}\ \hat{e}$	upTSM definition
			$\texttt{uimplicitp}[\hat{a}](\hat{e})$	$\texttt{implicit syntax}\ \hat{a}\ \texttt{for patterns in}\ \hat{e}$	upTSM designation
URule	$\hat{r}$	::=	$\texttt{urule}(\hat{p}.\hat{e})$	$\hat{p} \Rightarrow \hat{e}$	match rule
UPat	$\hat{p}$	::=	$\hat{x}$	$\hat{x}$	sigil pattern
			$\texttt{uwildp}$	$\_$	wildcard pattern
			$\texttt{ufoldp}(\hat{p})$	$\texttt{fold}(\hat{p})$	fold pattern
			$\texttt{utplp}[L](\{i \hookrightarrow \hat{p}_i\}_{i \in L})$	$\langle \{i \hookrightarrow \hat{p}_i\}_{i \in L} \rangle$	labeled tuple pattern
			$\texttt{uinjp}[\ell](\hat{p})$	$\texttt{inj}[\ell](\hat{p})$	injection pattern
			$\texttt{uapuptsm}[b][\hat{a}]$	$\hat{a}\ /b/$	upTSM application
			$\texttt{uplit}[b]$	$/b/$	upTSM unadorned literal

Figure 7.2: Abstract syntax of unexpanded types, expressions, rules and patterns in $\mathsf{miniVerse}^{\mathbf{B}}_{\mathbf{U}}$.

expansion. We **cannot** adopt the usual definition of α-renaming of identifiers, because unexpanded types and expressions are still in a "partially parsed" state – the literal bodies, b, within an unexpanded expression might contain spliced subterms that are "surfaced" by a TSM only during typed expansion, as we will detail below.

Each inner core form (defined in Figure 4.2) maps onto an outer surface form. In particular:

- Each type variable, t, maps onto a unique type sigil, written $\widehat{t}$.
- Each type form, τ, maps onto an unexpanded type form, $\mathcal{U}(\tau)$, according to the definition of $\mathcal{U}(\tau)$ in Sec. 3.2.5.
- Each expression variable, x, maps onto a unique expression sigil, written $\widehat{x}$.
- Each expanded expression form, e, maps onto an unexpanded expression form $\mathcal{U}(e)$ as follows:

$$\begin{aligned}
\mathcal{U}(x) &= \widehat{x} \\
\mathcal{U}(\texttt{lam\{}\tau\texttt{\}(}x.e\texttt{)}) &= \texttt{ulam\{}\mathcal{U}(\tau)\texttt{\}(}\widehat{x}.\mathcal{U}(e)\texttt{)} \\
\mathcal{U}(\texttt{ap(}e_1\texttt{;}e_2\texttt{)}) &= \texttt{uap(}\mathcal{U}(e_1)\texttt{;}\mathcal{U}(e_2)\texttt{)} \\
\mathcal{U}(\texttt{tlam(}t.e\texttt{)}) &= \texttt{utlam(}\widehat{t}.\mathcal{U}(e)\texttt{)} \\
\mathcal{U}(\texttt{tap\{}\tau\texttt{\}(}e\texttt{)}) &= \texttt{utap\{}\mathcal{U}(\tau)\texttt{\}(}\mathcal{U}(e)\texttt{)} \\
\mathcal{U}(\texttt{fold\{}t.\tau\texttt{\}(}e\texttt{)}) &= \texttt{uasc\{urec(}\widehat{t}.\mathcal{U}(\tau)\texttt{)\}(ufold(}\mathcal{U}(e)\texttt{))} \\
\mathcal{U}(\texttt{unfold(}e\texttt{)}) &= \texttt{uunfold(}\mathcal{U}(e)\texttt{)} \\
\mathcal{U}(\texttt{tpl}[L]\texttt{(\{}i \hookrightarrow e_i\texttt{\}}_{i\in L}\texttt{)}) &= \texttt{utpl\{}L\texttt{\}(\{}i \hookrightarrow \mathcal{U}(e_i)\texttt{\}}_{i\in L}\texttt{)} \\
\mathcal{U}(\texttt{inj}[L;\ell]\texttt{\{\{}i \hookrightarrow \tau_i\texttt{\}}_{i\in L}\texttt{\}(}e\texttt{)}) &= \texttt{uasc\{usum}[L]\texttt{(\{}i \hookrightarrow \mathcal{U}(\tau_i)\texttt{\}}_{i\in L}\texttt{)\}(uin[}\ell\texttt{](}\mathcal{U}(e)\texttt{))} \\
\mathcal{U}(\texttt{match}[n]\texttt{\{}\tau\texttt{\}(}e\texttt{;}\{r_i\}_{1\le i\le n}\texttt{)}) &= \texttt{uasc\{}\mathcal{U}(\tau)\texttt{\}(umatch}[n]\texttt{(}\mathcal{U}(e)\texttt{;}\{\mathcal{U}(r_i)\}_{1\le i\le n}\texttt{))}
\end{aligned}$$

  Notice that some type arguments that appear in e appear within a type ascription in $\mathcal{U}(e)$.

- The expanded rule form maps onto the unexpanded rule form as follows:

$$\mathcal{U}(\texttt{rule(}p.e\texttt{)}) = \texttt{urule(}\mathcal{U}(p).\mathcal{U}(e)\texttt{)}$$

- Each expanded pattern form, p, maps onto the unexpanded pattern form $\mathcal{U}(p)$ as follows:

$$\begin{aligned}
\mathcal{U}(x) &= \widehat{x} \\
\mathcal{U}(\texttt{wildp}) &= \texttt{uwildp} \\
\mathcal{U}(\texttt{foldp(}p\texttt{)}) &= \texttt{ufoldp(}\mathcal{U}(p)\texttt{)} \\
\mathcal{U}(\texttt{tplp}[L]\texttt{(\{}i \hookrightarrow p_i\texttt{\}}_{i\in L}\texttt{)}) &= \texttt{utplp}[L]\texttt{(\{}i \hookrightarrow \mathcal{U}(p_i)\texttt{\}}_{i\in L}\texttt{)} \\
\mathcal{U}(\texttt{injp}[\ell]\texttt{(}p\texttt{)}) &= \texttt{uinjp}[\ell]\texttt{(}\mathcal{U}(p)\texttt{)}
\end{aligned}$$

The forms related to TSM implicits are highlighted in gray in Figure 7.2.

7.2.3 Bidirectionally Typed Expansion

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

Judgement Form	Description
$\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \text{ type}$	$\hat{\tau}$ is well-formed and has expansion τ assuming $\hat{\Delta}$
$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau$	$\hat{e}$ has expansion e and synthesizes type τ under $\hat{\Psi}$ and $\hat{\Phi}$ assuming $\hat{\Delta}$ and $\hat{\Gamma}$
$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau$	$\hat{e}$ has expansion e when analyzed against type τ under $\hat{\Psi}$ and $\hat{\Phi}$ assuming $\hat{\Delta}$ and $\hat{\Gamma}$
$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \rightsquigarrow r \Rightarrow \tau \Mapsto \tau'$	$\hat{r}$ has expansion r and takes values of type τ to values of synthesized type τ' under $\hat{\Psi}$ and $\hat{\Phi}$ assuming $\hat{\Delta}$ and $\hat{\Gamma}$
$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \rightsquigarrow r \Leftarrow \tau \Mapsto \tau'$	$\hat{r}$ has expansion r and takes values of type τ to values of type τ' when $\tau's$ is provided for analysis under $\hat{\Psi}$ and $\hat{\Phi}$ assuming $\hat{\Delta}$ and $\hat{\Gamma}$
$\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\vert \hat{\Gamma}$	$\hat{p}$ has expansion p and type τ and generates hypotheses $\hat{\Gamma}$ under upTSM context $\hat{\Phi}$ assuming Δ

Type Expansion

Unexpanded type formation contexts, $\hat{\Delta}$, were defined in Sec. **??**. The *type expansion judgement*, $\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \text{ type}$, is inductively defined by Rules (B.5).

Typed Expression Expansion

In order to clearly define the semantics of TSM implicits, we must make a judgemental distinction between type synthesis and type analysis. In the latter, the type is presumed known, while in the former, it must be synthesized by examining the term that is the subject of the judgement. Expressions of unadorned literal form can only be analyzed against a known type.

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, are defined mutually inductively by Rules (7.1) and Rules (7.3), respectively, as follows.

Type Synthesis *Unexpanded typing contexts*, $\hat{\Gamma}$, were defined in Sec. 3.2.6. Sigils that appear in $\hat{\Gamma}$ have the expansion and synthesize the type that $\hat{\Gamma}$ assigns to them.

$$\frac{}{\hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{x} \rightsquigarrow x \Rightarrow \tau} \tag{7.1a}$$

A *type ascription* can be placed on an unexpanded expression to specify the type that it should be analyzed against. The ascribed type is synthesized if type analysis succeeds.

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \ \mathsf{type} \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uasc\{\hat{\tau}\}(\hat{e})} \rightsquigarrow e \Rightarrow \tau} \tag{7.1b}$$

We define let-binding of a value in synthetic position primitively in $\mathsf{miniVerse}^{\mathbf{B}}_{\mathbf{U}}$. The following rule governs such bindings in synthetic position.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \rightsquigarrow e' \Rightarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uletval}(\hat{e}; \hat{x}.\hat{e}') \rightsquigarrow \mathtt{ap}(\mathtt{lam\{\tau\}}(x.e'); e) \Rightarrow \tau'} \tag{7.1c}$$

Functions with an argument type annotation can appear in synthetic position.

$$\frac{\hat{\Delta} \vdash \hat{\tau}_1 \rightsquigarrow \tau_1 \ \mathsf{type} \qquad \hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau_1 \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau_2}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{ulam\{\hat{\tau}_1\}}(\hat{x}.\hat{e}) \rightsquigarrow \mathtt{lam\{\tau_1\}}(x.e) \Rightarrow \mathtt{parr}(\tau_1; \tau_2)} \tag{7.1d}$$

Function applications can appear in synthetic position. The argument is analyzed against the argument type synthesized by the function.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_1 \rightsquigarrow e_1 \Rightarrow \mathtt{parr}(\tau_2; \tau) \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_2 \rightsquigarrow e_2 \Leftarrow \tau_2}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uap}(\hat{e}_1; \hat{e}_2) \rightsquigarrow \mathtt{ap}(e_1; e_2) \Rightarrow \tau} \tag{7.1e}$$

Type lambdas and type applications can appear in synthetic position.

$$\frac{\hat{\Delta}, \hat{t} \rightsquigarrow t \ \mathsf{type}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{utlam}(\hat{t}.\hat{e}) \rightsquigarrow \mathtt{tlam}(t.e) \Rightarrow \mathtt{all}(t.\tau)} \tag{7.1f}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \mathtt{all}(t.\tau) \qquad \hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau' \ \mathsf{type}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{utap\{\hat{\tau}'\}}(\hat{e}) \rightsquigarrow \mathtt{tap\{\tau'\}}(e) \Rightarrow [\tau'/t]\tau} \tag{7.1g}$$

Unfoldings can appear in synthetic position.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \mathtt{rec}(t.\tau)}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uunfold}(\hat{e}) \rightsquigarrow \mathtt{unfold}(e) \Rightarrow [\mathtt{rec}(t.\tau)/t]\tau} \tag{7.1h}$$

Labeled tuples can appear in synthetic position. Each of the field values are then in synthetic position.

$$\frac{\{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_i \rightsquigarrow e_i \Rightarrow \tau_i\}_{i \in L}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{utpl\{L\}}(\{i \hookrightarrow \hat{e}_i\}_{i \in L}) \rightsquigarrow \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) \Rightarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{7.1i}$$

Fields can be projected out of a labeled tuple in synthetic position.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \mathtt{prod}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uprj}[\ell](\hat{e}) \rightsquigarrow \mathtt{prj}[\ell](e) \Rightarrow \tau} \tag{7.1j}$$

Match expressions can appear in synthetic position, as long as there is at least one rule.

$$\frac{n > 0 \qquad \hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \rightsquigarrow r_i \Rightarrow \tau \Mapsto \tau'\}_{1 \leq i \leq n}}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{umatch}[n](\hat{e}; \{\hat{r}_i\}_{1 \leq i \leq n}) \rightsquigarrow \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \leq i \leq n}) \Rightarrow \tau'} \tag{7.1k}$$

[resume] ueTSMs can be defined and applied in synthetic position.

$$\frac{\begin{array}{c}\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \varnothing\,\varnothing \vdash e_{\mathsf{parse}} : \mathtt{parr}(\mathtt{Body}; \mathtt{ParseResultSE}) \\ \hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{setsm}(\tau; e_{\mathsf{parse}}); \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau'\end{array}}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{for\ expressions}\ \{e_{\mathsf{parse}}\}\ \mathtt{in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \tag{7.2a}$$

$$\frac{\begin{array}{c}b \downarrow_{\mathsf{Body}} e_{\mathsf{body}} \qquad e_{\mathsf{parse}}(e_{\mathsf{body}}) \Downarrow \mathtt{inj}[\mathtt{Success}](e_{\mathsf{proto}}) \qquad e_{\mathsf{proto}} \uparrow_{\mathsf{ProtoExpr}} \grave{e} \\ \varnothing\,\varnothing \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{setsm}(\tau; e_{\mathsf{parse}}); \hat{\Phi}; b} \grave{e} \rightsquigarrow e \Leftarrow \tau\end{array}}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{setsm}(\tau; e_{\mathsf{parse}}); \hat{\Phi}} \hat{a}\ /b/ \rightsquigarrow e \Rightarrow \tau} \tag{7.2b}$$

These rules are nearly identical to Rules (**??**) and (**??**), differing only in that the typed expansion premises have been replaced by corresponding synthetic typed expansion premises. The premises of these rules can be understood as described in Sections 3.2.7 and 3.2.8. The body encoding judgement and candidate expansion expression decoding judgements were characterized in Sec. 4.2.5. We discuss candidate expansion validation in Sec. 7.2.5 below.

To support ueTSM implicits, ueTSM contexts, $\hat{\Psi}$, are redefined to take the form $\langle \mathcal{A}; \Psi; \mathcal{I} \rangle$. TSM naming contexts, $\mathcal{A}$, and ueTSM definition contexts, Ψ, were defined in Sec. 4.2.5. We write $\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{setsm}(\tau;\ e_{\mathsf{parse}})$ when $\hat{\Psi} = \langle \mathcal{A}; \Psi; \mathcal{I} \rangle$ as shorthand for

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

TSM designation contexts, $\mathcal{I}$, are finite functions that map each type $\tau \in \mathrm{dom}(\mathcal{I})$ to the *TSM designation* $\tau \hookrightarrow a$, for some symbol 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 TSM designation context in the ueTSM context is updated by expressions of ueTSM designation form. Such expressions can appear in synthetic position, where they are governed by the following rule:

$$\frac{\hat{\Delta}\,\hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \mathtt{setsm}(\tau; e_{\mathsf{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle; \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau'}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \mathtt{setsm}(\tau; e_{\mathsf{parse}}); \mathcal{I} \rangle; \hat{\Phi}} \mathtt{implicit\ syntax}\ \hat{a}\ \mathtt{for\ expressions\ in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \tag{7.2c}$$

Like ueTSMs, upTSMs can be defined in synthetic position.

$$\frac{\begin{array}{c}\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \varnothing\,\varnothing \vdash e_{\mathsf{parse}} : \mathtt{parr}(\mathtt{Body}; \mathtt{ParseResultSP}) \\ \hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}, \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{sptsm}(\tau; e_{\mathsf{parse}})} \hat{e} \rightsquigarrow e \Rightarrow \tau'\end{array}}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{for\ patterns\ by\ static}\ e_{\mathsf{parse}}\ \mathtt{end\ in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \tag{7.2d}$$

This rule is nearly identical to Rule (**??**), differing only in that the typed expansion premise has been replaced by the corresponding synthetic typed expansion premise. The premises can be understood as described in Section 4.2.6.

To support upTSM implicits, upTSM contexts, $\hat{\Phi}$, are redefined to take the form $\langle \mathcal{A}; \Phi; \mathcal{I} \rangle$. upTSM definition contexts, Φ, were defined in Sec. 4.2.6. We write $\hat{\Phi}, \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{sptsm}(\tau; e_{\text{parse}})$ when $\hat{\Phi} = \langle \mathcal{A}; \Phi; \mathcal{I} \rangle$ as shorthand for

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

The TSM designation context in the upTSM context is updated by expressions of upTSM designation form. Such expressions can appear in synthetic position, where they are governed by the following rule:

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \mathtt{sptsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle} \hat{e} \rightsquigarrow e \Rightarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \mathtt{setsm}(\tau; e_{\text{parse}}); \mathcal{I} \rangle} \mathtt{implicit\ syntax}\ \hat{a}\ \mathtt{for\ patterns\ in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \tag{7.2e}$$

Type Analysis Type analysis subsumes type synthesis, in that when a type can be synthesized for an unexpanded expression, that unexpanded expression can also be analyzed against that type, producing the same expansion. This is expressed by the following *subsumption rule* for unexpanded expressions.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau} \tag{7.3a}$$

Additional rules are needed for certain forms in order to propagate types for analysis into subexpressions, and for forms that can appear only in analytic position.

Rule (7.1c) governed value bindings in synthetic position. The following rule governs value bindings in analytic position.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e}' \rightsquigarrow e' \Leftarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \mathtt{uletval}(\hat{e}; \hat{x}.\hat{e}') \rightsquigarrow \mathtt{ap}(\mathtt{lam}\{\tau\}(x.e'); e) \Leftarrow \tau'} \tag{7.3b}$$

An unannotated function can appear only in analytic position. The argument type is determined from the type that the unannotated function is being analyzed against.

$$\frac{\hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau_1 \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau_2}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \mathtt{uanalam}(\hat{x}.\hat{e}) \rightsquigarrow \mathtt{lam}\{\tau_1\}(x.e) \Leftarrow \mathtt{parr}(\tau_1; \tau_2)} \tag{7.3c}$$

Rule (7.1f) governed type lambdas in synthetic position. The following rule governs type lambdas in analytic position.

$$\frac{\hat{\Delta}, \hat{t} \rightsquigarrow t\ \mathtt{type}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \mathtt{utlam}(\hat{t}.\hat{e}) \rightsquigarrow \mathtt{tlam}(t.e) \Leftarrow \mathtt{all}(t.\tau)} \tag{7.3d}$$

Values of recursive types can be introduced only in analytic position.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow [\mathtt{rec}(t.\tau)/t]\tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{ufold}(\hat{e}) \rightsquigarrow \mathtt{fold}\{t.\tau\}(e) \Leftarrow \mathtt{rec}(t.\tau)} \tag{7.3e}$$

Rule (7.1i) governed labeled tuples in synthetic position. The following rule governs labeled tuples in analytic position.

$$\frac{\{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_i \rightsquigarrow e_i \Leftarrow \tau_i\}_{i \in L}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{utpl}\{L\}(\{i \hookrightarrow \hat{e}_i\}_{i \in L}) \rightsquigarrow \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) \Leftarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{7.3f}$$

Values of labeled sum type can appear only in analytic position.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\left(\begin{array}{c}\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uin}[\ell](\hat{e}) \\ \rightsquigarrow \\ \mathtt{inj}[L,\ell;\ell]\{\{i \hookrightarrow \tau_i\}_{i \in L};\ell \hookrightarrow \tau\}(e) \Leftarrow \mathtt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i \in L};\ell \hookrightarrow \tau)\end{array}\right)} \tag{7.3g}$$

Rule (7.1k) governed match expressions in synthetic position. The following rule governs match expressions in analytic position.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \rightsquigarrow r_i \Leftarrow \tau \Mapsto \tau'\}_{1 \le i \le n}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{umatch}[n](\hat{e};\{\hat{r}_i\}_{1 \le i \le n}) \rightsquigarrow \mathtt{match}[n]\{\tau'\}(e;\{r_i\}_{1 \le i \le n}) \Leftarrow \tau'} \tag{7.3h}$$

[resume] Rule (7.2a) governed ueTSM definitions in synthetic position. The following rule governs ueTSM definitions in analytic position.

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \varnothing\,\varnothing \vdash e_{\mathrm{parse}} : \mathtt{parr}(\mathtt{Body};\mathtt{ParseResultSE}) \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi},\hat{a} \rightsquigarrow a \hookrightarrow \mathtt{setsm}(\tau;e_{\mathrm{parse}});\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{for\ expressions}\ \{e_{\mathrm{parse}}\}\ \mathtt{in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{7.4a}$$

Rule (7.2c) governed ueTSM designations in synthetic position. The following rule governs ueTSM designations in analytic position.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a;\Psi,a \hookrightarrow \mathtt{setsm}(\tau;e_{\mathrm{parse}});\mathcal{I} \uplus \tau \hookrightarrow a\rangle;\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a;\Psi,a \hookrightarrow \mathtt{setsm}(\tau;e_{\mathrm{parse}});\mathcal{I}\rangle;\hat{\Phi}} \mathtt{implicit\ syntax}\ \hat{a}\ \mathtt{for\ expressions\ in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{7.4b}$$

An expression of unadorned literal form can appear only in analytic position. The following rule extracts the TSM designated at the type that the expression is being analyzed against from the TSM designation context in the ueTSM context and applies it implicitly, i.e. the premises correspond to those of Rule (7.2b).

$$\frac{b \downarrow_{\mathsf{Body}} e_{\mathrm{body}} \qquad e_{\mathrm{parse}}(e_{\mathrm{body}}) \Downarrow \mathtt{inj}[\mathtt{Success}](e_{\mathrm{proto}}) \qquad e_{\mathrm{proto}} \uparrow_{\mathsf{ProtoExpr}} \grave{e} \qquad \varnothing\,\varnothing \vdash^{\hat{\Delta};\hat{\Gamma};\langle \mathcal{A};\Psi,a \hookrightarrow \mathtt{setsm}(\tau;e_{\mathrm{parse}});\mathcal{I} \uplus \tau \hookrightarrow a\rangle;\hat{\Phi};b} \grave{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A};\Psi,a \hookrightarrow \mathtt{setsm}(\tau;e_{\mathrm{parse}});\mathcal{I} \uplus \tau \hookrightarrow a\rangle;\hat{\Phi}} \mathtt{uelit}[b] \rightsquigarrow e \Leftarrow \tau} \tag{7.4c}$$

Rule (7.2d) governed upTSM definitions in synthetic position. The following rule governs upTSM definitions in analytic position.

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \text{ type} \qquad \emptyset\,\emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResultSP)} \qquad \hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{sptsm}(\tau;e_{\text{parse}})} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for patterns by static}\ e_{\text{parse}}\ \texttt{end in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{7.4d}$$

Rule (7.2e) governed upTSM designations in synthetic position. The following rule governs upTSM designations in analytic position.

$$\frac{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\langle \mathcal{A}\uplus\hat{a}\rightsquigarrow a;\Phi,a\hookrightarrow \texttt{sptsm}(\tau;e_{\text{parse}});\mathcal{I}\uplus\tau\hookrightarrow a\rangle} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\langle \mathcal{A}\uplus\hat{a}\rightsquigarrow a;\Phi,a\hookrightarrow \texttt{setsm}(\tau;e_{\text{parse}});\mathcal{I}\rangle} \texttt{implicit syntax}\ \hat{a}\ \texttt{for patterns in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{7.4e}$$

Typed Rule Expansion

The synthetic typed rule expansion judgement is invoked iteratively by Rule (7.1k) to synthesize a type, τ', from the branch expressions in the rule sequence. This judgement is defined mutually inductively with Rules (7.1) and Rules (7.3) by the following rule.

$$\frac{\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \langle \mathcal{G}';\Gamma'\rangle \qquad \langle \mathcal{D};\Delta\rangle\ \langle \mathcal{G}\uplus\mathcal{G}';\Gamma\cup\Gamma'\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau'}{\langle \mathcal{D};\Delta\rangle\ \langle \mathcal{G};\Gamma\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{urule}(\hat{p}.\hat{e}) \rightsquigarrow \texttt{rule}(p.e) \Rightarrow \tau \Mapsto \tau'} \tag{7.5}$$

The analytic typed rule expansion judgement is invoked iteratively by Rule (7.3h). This judgement is defined mutually inductively with Rules (7.1), Rules (7.3), and Rule (7.5) by the following rule, which is the analytic analag of Rule (7.5).

$$\frac{\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \langle \mathcal{G}';\Gamma'\rangle \qquad \langle \mathcal{D};\Delta\rangle\ \langle \mathcal{G}\uplus\mathcal{G}';\Gamma\cup\Gamma'\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\langle \mathcal{D};\Delta\rangle\ \langle \mathcal{G};\Gamma\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{urule}(\hat{p}.\hat{e}) \rightsquigarrow \texttt{rule}(p.e) \Leftarrow \tau \Mapsto \tau'} \tag{7.6}$$

The premises of these rules can be understood as described in Sec. 4.2.5.

Typed Pattern Expansion

The typed pattern expansion judgement is inductively defined by Rules (7.7) as follows.

The following rules are written identically to the typed pattern expansion rules for shared pattern forms in miniVerse$_{\mathbf{S}}$, i.e. Rules (B.8a) through (B.8e).

$$\frac{}{\Delta \vdash_{\hat{\Phi}} \hat{x} \rightsquigarrow x : \tau \dashv\!\mid \langle \hat{x} \rightsquigarrow x; x : \tau\rangle} \tag{7.7a}$$

$$\frac{}{\Delta \vdash_{\hat{\Phi}} \texttt{uwildp} \rightsquigarrow \texttt{wildp} : \tau \dashv\!\mid \langle \emptyset;\emptyset\rangle} \tag{7.7b}$$

$$\frac{\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : [\texttt{rec}(t.\tau)/t]\tau \dashv\!\mid \hat{\Gamma}}{\Delta \vdash_{\hat{\Phi}} \texttt{ufoldp}(\hat{p}) \rightsquigarrow \texttt{foldp}(p) : \texttt{rec}(t.\tau) \dashv\!\mid \hat{\Gamma}} \tag{7.7c}$$

$$\frac{\{\Delta \vdash_{\hat{\Phi}} \hat{p}_i \rightsquigarrow p_i : \tau_i \dashv\!\mid \hat{\Gamma}_i\}_{i \in L}}{\left(\begin{array}{c}\Delta \vdash_{\hat{\Phi}} \texttt{utplp}[L](\{i \hookrightarrow \hat{p}_i\}_{i \in L}) \\ \rightsquigarrow \\ \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\!\mid \cup_{i \in L} \hat{\Gamma}_i\end{array}\right)} \tag{7.7d}$$

$$\frac{\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \hat{\Gamma}}{\Delta \vdash_{\hat{\Phi}} \texttt{uinjp}[\ell](\hat{p}) \rightsquigarrow \texttt{injp}[\ell](p) : \texttt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\!\mid \hat{\Gamma}} \tag{7.7e}$$

The following rule governs upTSM application. It is written identically to Rule (B.8f).

$$\frac{\begin{array}{c}b \downarrow_{\mathsf{Body}} e_{\mathsf{body}} \qquad e_{\mathsf{parse}}(e_{\mathsf{body}}) \Downarrow \texttt{inj}[\texttt{Success}](e_{\mathsf{proto}}) \qquad e_{\mathsf{proto}} \uparrow_{\mathsf{ProtoPat}} \hat{p} \\ \vdash^{\Delta;\hat{\Phi},\hat{a} \rightsquigarrow a \hookrightarrow \texttt{sptsm}(\tau;e_{\mathsf{parse}});b} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid^{\hat{\Gamma}}\end{array}}{\Delta \vdash_{\hat{\Phi},\hat{a} \rightsquigarrow a \hookrightarrow \texttt{sptsm}(\tau;e_{\mathsf{parse}})} \hat{a}\ /b/ \rightsquigarrow p : \tau \dashv\!\mid \hat{\Gamma}} \tag{7.7f}$$

Unexpanded patterns of unadorned literal form are governed by the following rule, which extracts the designated upTSM from the upTSM context and applies it implicitly, i.e. the premises correspond to those of Rule (7.7f).

$$\frac{\begin{array}{c}b \downarrow_{\mathsf{Body}} e_{\mathsf{body}} \qquad e_{\mathsf{parse}}(e_{\mathsf{body}}) \Downarrow \texttt{inj}[\texttt{Success}](e_{\mathsf{proto}}) \qquad e_{\mathsf{proto}} \uparrow_{\mathsf{ProtoPat}} \hat{p} \\ \vdash^{\Delta;\langle \mathcal{A};\Phi,a \hookrightarrow \texttt{sptsm}(\tau;e_{\mathsf{parse}});\mathcal{I},\tau \hookrightarrow a\rangle;b} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid^{\hat{\Gamma}}\end{array}}{\Delta \vdash_{\langle \mathcal{A};\Phi,a \hookrightarrow \texttt{sptsm}(\tau;e_{\mathsf{parse}});\mathcal{I},\tau \hookrightarrow a\rangle} /b/ \rightsquigarrow p : \tau \dashv\!\mid \hat{\Gamma}} \tag{7.7g}$$

Outer Surface Expressibility

The following lemma establishes that each well-typed expanded pattern can be expressed as an unexpanded pattern matching values of the same type and generating the same hypotheses and corresponding sigil updates. The metafunction $\mathcal{U}(\Gamma)$ was defined in 4.2.5.

Lemma 7.2 (Pattern Expressibility)**.** *If* $\Delta \vdash p : \tau \dashv\!\mid \Gamma$ *then* $\Delta \vdash_{\hat{\Phi}} \mathcal{U}(p) \rightsquigarrow p : \tau \dashv\!\mid \mathcal{U}(\Gamma)$.

Proof. By rule induction over Rules (B.4), using the definitions of $\mathcal{U}(\Gamma)$ and $\mathcal{U}(p)$. In each case, we can apply the IH to or over each premise, then apply the corresponding rule in Rules (7.7). □

We can now establish the Expressibility Theorem – that each well-typed expanded expression, e, can be expressed as an unexpanded expression, $\hat{e}$, which synthesizes the same type under the corresponding contexts.

Theorem 7.3 (Expressibility)**.** *Both of the following hold:*

1. *If* $\Delta\ \Gamma \vdash e : \tau$ *then* $\mathcal{U}(\Delta)\ \mathcal{U}(\Gamma) \vdash_{\hat{\Psi};\hat{\Phi}} \mathcal{U}(e) \rightsquigarrow e \Rightarrow \tau$.
2. *If* $\Delta\ \Gamma \vdash r : \tau \Mapsto \tau'$ *then* $\mathcal{U}(\Delta)\ \mathcal{U}(\Gamma) \vdash_{\hat{\Psi};\hat{\Phi}} \mathcal{U}(r) \rightsquigarrow r \Rightarrow \tau \Mapsto \tau'$.

Proof. By mutual rule induction over Rules (B.2) and Rule (B.3) using the definitions of $\mathcal{U}(\Delta), \mathcal{U}(\Gamma), \mathcal{U}(e)$ and $\mathcal{U}(r)$. In each case, we apply the IH, part 1 to or over each typing premise, the IH, part 2 over each rule typing premise, Lemma **??** to or over each type formation premise, Lemma 7.2 to each pattern typing premise, then derive the conclusion by applying Rules (7.1) and Rule (7.5). □

Sort			Operational Form	Stylized Form	Description
CETyp	$\grave{\tau}$	::=	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
			$\texttt{splicedt}[m;n]$	$\texttt{splicedt}\langle m,n\rangle$	spliced
CEExp	$\grave{e}$	::=	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
			$\texttt{pranalam}(x.\grave{e})$	$\lambda x.\grave{e}$	abstraction (unannotated)
			$\texttt{prlam}\{\grave{\tau}\}(x.\grave{e})$	$\lambda x{:}\grave{\tau}.\grave{e}$	abstraction (annotated)
			$\texttt{prap}(\grave{e};\grave{e})$	$\grave{e}(\grave{e})$	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
			$\texttt{prin}[\ell](\grave{e})$	$\texttt{inj}[\ell](\grave{e})$	injection
			$\texttt{prmatch}[n](\grave{e};\{\hat{r}_i\}_{1\le i\le n})$	$\texttt{match}\ \grave{e}\ \{\hat{r}_i\}_{1\le i\le n}$	match
			$\texttt{splicede}[m;n]$	$\texttt{splicede}\langle m,n\rangle$	spliced
CERule	$\hat{r}$	::=	$\texttt{prrule}(p.\grave{e})$	$p \Rightarrow \grave{e}$	rule
CEPat	$\grave{p}$	::=	$\texttt{prwildp}$	$\_$	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
			$\texttt{splicedp}[m;n]$	$\texttt{splicedp}\langle m,n\rangle$	spliced

Figure 7.3: Abstract syntax of candidate expansion types, expressions, rules and patterns in $\mathsf{miniVerse}^{\mathbf{B}}_{\mathbf{U}}$. Candidate expansion terms are identified up to α-equivalence.

7.2.4 Syntax of Candidate Expansions

Figure 7.3 defines the syntax of candidate expansion types (or *ce-types*), $\grave{\tau}$, candidate expansion expressions (or *ce-expressions*), $\grave{e}$, candidate expansion rules (or *ce-rules*), $\grave{r}$, and candidate expansion patterns (or *ce-patterns*), $\grave{p}$. Candidate expansion terms are identified up to α-equivalence in the usual manner.

Each inner core form, except for the variable pattern form, maps onto a candidate expansion form. In particular:

- Each type form maps onto a ce-type form according to the metafunction $\mathcal{P}(\tau)$, defined in Sec. 3.2.9.
- Each expanded expression form maps onto a ce-expression form according to the metafunction $\mathcal{P}(e)$, defined as follows:

$$
\begin{aligned}
\mathcal{P}(x) &= x \\
\mathcal{P}(\texttt{lam\{}\tau\texttt{\}(}x.e\texttt{)}) &= \texttt{prlam\{}\mathcal{P}(\tau)\texttt{\}(}x.\mathcal{P}(e)\texttt{)} \\
\mathcal{P}(\texttt{ap(}e_1\texttt{;}e_2\texttt{)}) &= \texttt{prap(}\mathcal{P}(e_1)\texttt{;}\mathcal{P}(e_2)\texttt{)} \\
\mathcal{P}(\texttt{tlam(}t.e\texttt{)}) &= \texttt{prtlam(}t.\mathcal{P}(e)\texttt{)} \\
\mathcal{P}(\texttt{tap\{}\tau\texttt{\}(}e\texttt{)}) &= \texttt{prtap\{}\mathcal{P}(\tau)\texttt{\}(}\mathcal{P}(e)\texttt{)} \\
\mathcal{P}(\texttt{fold\{}t.\tau\texttt{\}(}e\texttt{)}) &= \texttt{prasc\{prrec(}t.\mathcal{P}(\tau)\texttt{)\}(prfold(}\mathcal{P}(e)\texttt{))} \\
\mathcal{P}(\texttt{unfold(}e\texttt{)}) &= \texttt{prunfold(}\mathcal{P}(e)\texttt{)} \\
\mathcal{P}(\texttt{tpl}[L]\texttt{(}\{i \hookrightarrow e_i\}_{i\in L}\texttt{)}) &= \texttt{prtpl\{}L\texttt{\}(}\{i \hookrightarrow \mathcal{P}(e_i)\}_{i\in L}\texttt{)} \\
\mathcal{P}(\texttt{inj}[L;\ell]\texttt{\{}\{i \hookrightarrow \tau_i\}_{i\in L}\texttt{\}(}e\texttt{)}) &= \texttt{prasc\{prsum}[L]\texttt{(}\{i \hookrightarrow \mathcal{P}(\tau_i)\}_{i\in L}\texttt{)\}(prin}[\ell]\texttt{(}\mathcal{P}(e)\texttt{))} \\
\mathcal{P}(\texttt{match}[n]\texttt{\{}\tau\texttt{\}(}e\texttt{;}\{r_i\}_{1\le i\le n}\texttt{)}) &= \texttt{prasc\{}\mathcal{P}(\tau)\texttt{\}(prmatch}[n]\texttt{(}\mathcal{P}(e)\texttt{;}\{\mathcal{P}(r_i)\}_{1\le i\le n}\texttt{))}
\end{aligned}
$$

- The expanded rule form maps onto the ce-rule form according to the metafunction $\mathcal{P}(r)$, defined as follows:

$$\mathcal{P}(\texttt{rule(}p.e\texttt{)}) = \texttt{prrule(}p.\mathcal{P}(e)\texttt{)}$$

- Each expanded pattern form, except for the variable pattern form, maps onto a ce-pattern form according to the metafunction $\mathcal{P}(p)$, defined in Sec. 4.2.8.

There are three other candidate expansion forms: a ce-type form for *references to spliced unexpanded types*, $\texttt{splicedt}[m;n]$, a ce-expression form for *references to spliced unexpanded expressions*, $\texttt{splicede}[m;n]$, and a ce-pattern form for *references to spliced unexpanded patterns*, $\texttt{splicedp}[m;n]$.

7.2.5 Bidirectional Candidate Expansion Validation

The *bidirectional candidate expansion validation judgements* validate ce-terms and simultaneously generate their final expansions.

Judgement Form	Description
$\Delta \vdash^{\mathbb{T}} \grave{\tau} \rightsquigarrow \tau$ type	$\grave{\tau}$ is well-formed and has expansion τ assuming Δ and type splicing scene $\mathbb{T}$
$\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau$	$\grave{e}$ has expansion e and synthesizes type τ assuming Δ and Γ and expression splicing scene $\mathbb{E}$
$\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau$	$\grave{e}$ has expansion e when analyzed against type τ assuming Δ and Γ and expression splicing scene $\mathbb{E}$
$\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r} \rightsquigarrow r \Rightarrow \tau \Mapsto \tau'$	$\grave{r}$ has expansion r and takes values of type τ to values of synthesized type τ' assuming Δ and Γ and $\mathbb{E}$
$\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r} \rightsquigarrow r \Leftarrow \tau \Mapsto \tau'$	$\grave{r}$ has expansion r and takes values of type τ to values of type τ' when τ' is provided for analysis assuming Δ and Γ and $\mathbb{E}$
$\vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : \tau \dashv\mid^{\hat{\Gamma}}$	$\grave{p}$ expands to p and matches values of type τ generating assumptions $\hat{\Gamma}$ assuming pattern splicing scene $\mathbb{P}$

Expression splicing scenes, $\mathbb{E}$, are of the form $\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; \hat{\Phi}; b$, *type splicing scenes*, $\mathbb{T}$, are of the form $\hat{\Delta}; b$, and *pattern splicing scenes*, $\mathbb{P}$, are of the form $\Delta; \hat{\Phi}; b$. Their purpose is to "remember", during candidate expansion validation, the contexts, TSM environments and literal bodies from the TSM application site (cf. Rules (**??**) 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 the unexpanded typing context and TSM environments from $\mathbb{E}$:

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

Candidate Expansion Type Validation

The *ce-type validation judgement*, $\Delta \vdash^{\mathbb{T}} \grave{\tau} \rightsquigarrow \tau$ type, is inductively defined by Rules (B.9), which were defined in Sec. 3.2.10.

Bidirectional Candidate Expansion Expression Validation

Like the bidirectionally typed expression expansion judgements, the bidirectional ce-expression validation judgements distinguish type synthesis from type analysis. The *synthetic ce-expression validation judgement*, $\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau$, and the *analytic ce-expression validation judgement*, $\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau$, are defined mutually inductively with Rules (7.1) and Rules (7.3) by Rules (7.8) and Rules (7.9), respectively, as follows.

Type Synthesis Synthetic ce-expression validation is governed by the following rules.

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash^{\mathbb{E}} x \rightsquigarrow x \Rightarrow \tau} \tag{7.8a}$$

$$\frac{\Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau} \rightsquigarrow \tau \ \mathsf{type} \qquad \Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prasc}\{\grave{\tau}\}(\grave{e}) \rightsquigarrow e \Rightarrow \tau} \tag{7.8b}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \Delta\ \Gamma, x : \tau \vdash^{\mathbb{E}} \grave{e}' \rightsquigarrow e' \Rightarrow \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prletval}(\grave{e}; x.\grave{e}') \rightsquigarrow \mathtt{ap}(\mathtt{lam}\{\tau\}(x.e'); e) \Rightarrow \tau'} \tag{7.8c}$$

$$\frac{\Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau}_1 \rightsquigarrow \tau_1 \ \mathsf{type} \qquad \Delta\ \Gamma, x : \tau_1 \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau_2}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prlam}\{\grave{\tau}_1\}(x.\grave{e}) \rightsquigarrow \mathtt{lam}\{\tau_1\}(x.e) \Rightarrow \mathtt{parr}(\tau_1; \tau_2)} \tag{7.8d}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e}_1 \rightsquigarrow e_1 \Rightarrow \mathtt{parr}(\tau_2; \tau) \qquad \Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e}_2 \rightsquigarrow e_2 \Leftarrow \tau_2}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prap}(\grave{e}_1; \grave{e}_2) \rightsquigarrow \mathtt{ap}(e_1; e_2) \Rightarrow \tau} \tag{7.8e}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prtlam}(t.\grave{e}) \rightsquigarrow \mathtt{tlam}(t.e) \Rightarrow \mathtt{all}(t.\tau)} \tag{7.8f}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \mathtt{all}(t.\tau) \qquad \Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau}' \rightsquigarrow \tau' \ \mathsf{type}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prtap}\{\grave{\tau}'\}(\grave{e}) \rightsquigarrow \mathtt{tap}\{\tau'\}(e) \Rightarrow [\tau'/t]\tau} \tag{7.8g}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \mathtt{rec}(t.\tau)}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prunfold}(\grave{e}) \rightsquigarrow \mathtt{unfold}(e) \Rightarrow [\mathtt{rec}(t.\tau)/t]\tau} \tag{7.8h}$$

$$\frac{\{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e}_i \rightsquigarrow e_i \Rightarrow \tau_i\}_{i \in L}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prtpl}\{L\}(\{i \hookrightarrow \grave{e}_i\}_{i \in L}) \rightsquigarrow \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) \Rightarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{7.8i}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \mathtt{prod}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prprj}[\ell](\grave{e}) \rightsquigarrow \mathtt{prj}[\ell](e) \Rightarrow \tau} \tag{7.8j}$$

$$\frac{n > 0 \qquad \Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r}_i \rightsquigarrow r_i \Rightarrow \tau \Rightarrow \tau'\}_{1 \leq i \leq n}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prmatch}[n](\grave{e}; \{\grave{r}_i\}_{1 \leq i \leq n}) \rightsquigarrow \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \leq i \leq n}) \Rightarrow \tau'} \tag{7.8k}$$

$$\frac{\begin{array}{c}\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e} \qquad \langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle\ \langle \mathcal{G}; \Gamma_{\mathsf{app}} \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \\ \Delta \cap \Delta_{\mathsf{app}} = \emptyset \qquad \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma_{\mathsf{app}}) = \emptyset\end{array}}{\Delta\ \Gamma \vdash^{\langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle; \langle \mathcal{G}; \Gamma_{\mathsf{app}} \rangle; \hat{\Psi}; \hat{\Phi}; b} \mathtt{splicede}[m; n] \rightsquigarrow e \Rightarrow \tau} \tag{7.8l}$$

Rules (7.8a) through (7.8k) are analagous to Rules (7.1a) through (7.1k). Rule (7.8l) governs references to spliced unexpanded expressions in synthetic position, and can be understood as described in Sec. 3.2.10.

Type Analysis Analytic ce-expression validation is governed by the following rules.

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau} \tag{7.9a}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \Delta\ \Gamma, x : \tau \vdash^{\mathbb{E}} \grave{e}' \rightsquigarrow e' \Leftarrow \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prletval}(\grave{e}; x.\grave{e}') \rightsquigarrow \mathtt{ap}(\mathtt{lam}\{\tau\}(x.e'); e) \Leftarrow \tau'} \tag{7.9b}$$

$$\frac{\Delta\ \Gamma, x : \tau_1 \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau_2}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{pranalam}(x.\hat{e}) \rightsquigarrow \mathtt{lam}\{\tau_1\}(x.e) \Leftarrow \mathtt{parr}(\tau_1; \tau_2)} \tag{7.9c}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prtlam}(t.\grave{e}) \rightsquigarrow \mathtt{tlam}(t.e) \Leftarrow \mathtt{all}(t.\tau)} \tag{7.9d}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow [\mathtt{rec}(t.\tau)/t]\tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prfold}(\grave{e}) \rightsquigarrow \mathtt{fold}\{t.\tau\}(e) \Leftarrow \mathtt{rec}(t.\tau)} \tag{7.9e}$$

$$\frac{\{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e}_i \rightsquigarrow e_i \Leftarrow \tau_i\}_{i \in L}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prtpl}\{L\}(\{i \hookrightarrow \grave{e}_i\}_{i \in L}) \rightsquigarrow \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) \Leftarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{7.9f}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau}{\left(\begin{array}{c}\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prin}[\ell](\grave{e}) \\ \rightsquigarrow \\ \mathtt{inj}[L, \ell; \ell]\{\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau\}(e) \Leftarrow \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)\end{array}\right)} \tag{7.9g}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r}_i \rightsquigarrow r_i \Leftarrow \tau \Mapsto \tau'\}_{1 \leq i \leq n}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prmatch}[n](\grave{e}; \{\grave{r}_i\}_{1 \leq i \leq n}) \rightsquigarrow \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \leq i \leq n}) \Leftarrow \tau'} \tag{7.9h}$$

$$\frac{\begin{array}{c}\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e} \qquad \langle \mathcal{D}; \Delta_{\mathrm{app}} \rangle\ \langle \mathcal{G}; \Gamma_{\mathrm{app}} \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau \\ \Delta \cap \Delta_{\mathrm{app}} = \emptyset \qquad \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma_{\mathrm{app}}) = \emptyset\end{array}}{\Delta\ \Gamma \vdash^{\langle \mathcal{D}; \Delta_{\mathrm{app}} \rangle; \langle \mathcal{G}; \Gamma_{\mathrm{app}} \rangle; \hat{\Psi}; \hat{\Phi}; b} \mathtt{splicede}[m; n] \rightsquigarrow e \Leftarrow \tau} \tag{7.9i}$$

Rules (7.9a) through (7.9h) are analagous to Rules (7.3a) through (7.3h). Rule (7.9i) governs references to spliced unexpanded expressions in analytic position.

Bidirectional Candidate Expansion Rule Validation

The *synthetic ce-rule validation judgement* is defined mutually inductively with Rules (7.1) by the following rule.

$$\frac{\Delta \vdash p : \tau \dashv\!\mid \Gamma \qquad \Delta\ \Gamma \cup \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prrule}(p.\grave{e}) \rightsquigarrow \mathtt{rule}(p.e) \Rightarrow \tau \Mapsto \tau'} \tag{7.10}$$

The *analytic ce-rule validation judgement* is defined mutually inductively with Rules (7.3) by the following rule.

$$\frac{\Delta \vdash p : \tau \dashv\Vdash \Gamma \qquad \Delta\ \Gamma \cup \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prrule}(p.\grave{e}) \rightsquigarrow \mathtt{rule}(p.e) \Leftarrow \tau \Rightarrow \tau'} \tag{7.11}$$

Candidate Expansion Pattern Validation

The *ce-pattern validation judgement* is inductively defined by the following rules, which are written identically to Rules (B.12).

$$\frac{}{\vdash^{\mathbb{P}} \mathtt{prwildp} \rightsquigarrow \mathtt{wildp} : \tau \dashv\Vdash^{\langle \emptyset;\emptyset \rangle}} \tag{7.12a}$$

$$\frac{\vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : [\mathtt{rec}(t.\tau)/t]\tau \dashv\Vdash^{\hat{\Gamma}}}{\vdash^{\mathbb{P}} \mathtt{prfoldp}(\grave{p}) \rightsquigarrow \mathtt{foldp}(p) : \mathtt{rec}(t.\tau) \dashv\Vdash^{\hat{\Gamma}}} \tag{7.12b}$$

$$\frac{\{\vdash^{\mathbb{P}} \grave{p}_i \rightsquigarrow p_i : \tau_i \dashv\Vdash^{\hat{\Gamma}_i}\}_{i \in L}}{\left(\begin{array}{c} \vdash^{\mathbb{P}} \mathtt{prtplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i \in L}) \\ \rightsquigarrow \\ \mathtt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\Vdash^{\cup_{i \in L} \hat{\Gamma}_i} \end{array}\right)} \tag{7.12c}$$

$$\frac{\vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : \tau \dashv\Vdash^{\hat{\Gamma}}}{\vdash^{\mathbb{P}} \mathtt{prinjp}[\ell](\grave{p}) \rightsquigarrow \mathtt{injp}[\ell](p) : \mathtt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\Vdash^{\hat{\Gamma}}} \tag{7.12d}$$

$$\frac{\mathsf{parseUPat}(\mathsf{subseq}(b;m;n)) = \hat{p} \qquad \Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\Vdash \hat{\Gamma}}{\vdash^{\Delta;\hat{\Phi};b} \mathtt{splicedp}[m;n] \rightsquigarrow p : \tau \dashv\Vdash^{\hat{\Gamma}}} \tag{7.12e}$$

Candidate Expansion Expressibility

The following lemma establishes that each well-typed expanded expression, e, can be expressed as a valid ce-expression, $\mathcal{P}(e)$, that synthesizes the same type under the same contexts and any expression splicing scene.

Theorem 7.4 (Candidate Expansion Expression Expressibility)**.** *Both of the following hold:*

1. *If* $\Delta\ \Gamma \vdash e : \tau$ *then* $\Delta\ \Gamma \vdash^{\mathbb{E}} \mathcal{P}(e) \rightsquigarrow e \Rightarrow \tau$.
2. *If* $\Delta\ \Gamma \vdash r : \tau \Rightarrow \tau'$ *then* $\Delta\ \Gamma \vdash^{\mathbb{E}} \mathcal{P}(r) \rightsquigarrow r \Rightarrow \tau \Rightarrow \tau'$.

Proof. By mutual rule induction over Rules (B.2) and Rule (B.3). In each case, we apply the IH, part 1 to or over each typing premise, the IH, part 2 over each rule typing premise, Lemma **??** to or over each type formation premise and then derive the conclusion by applying Rules (7.8) and Rule (7.10) as needed. □

The following lemma establishes that every well-typed expanded pattern that generates no hypotheses can be expressed as a ce-pattern.

Lemma 7.5 (Candidate Expansion Pattern Expressibility). *If* $\Delta \vdash p : \tau \dashv\Vdash \varnothing$ *then* $\vdash^{\Delta;\hat{\Phi};b} \mathcal{P}(p) \rightsquigarrow p : \tau \dashv\Vdash^{\langle\varnothing;\varnothing\rangle}$.

Proof. The proof is nearly identical to the proof of Lemma **??**, differing only in that each mention of a rule in Rules (B.12) is replaced by a mention of the corresponding rule in Rules (7.12). □

7.2.6 Metatheory

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 candidate expansion patterns, because the judgements are mutually defined.

Theorem 7.6 (Typed Pattern Expansion). *Both of the following hold:*

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

Proof. My mutual rule induction over Rules (7.7) and Rules (7.12).

1. We induct on the premise. In the following, let $\hat{\Gamma} = \langle\mathcal{G};\Gamma\rangle$ and $\hat{\Phi} = \langle\mathcal{A};\Phi;\mathcal{I}\rangle$.

   Case (7.7a) through (7.7f). In each of these cases, the proof is written identically to the proof of the corresponding case in the proof of Theorem B.28.

   Case (7.7g). We have:

   (1) $\hat{p} = \texttt{uplit}[b]$ — 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) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}[\texttt{Success}](e_{\text{proto}})$ — by assumption
   (6) $e_{\text{proto}} \uparrow_{\mathsf{ProtoPat}} \grave{p}$ — by assumption
   (7) $\vdash^{\Delta;\langle\mathcal{A};\Phi',a\hookrightarrow\texttt{sptsm}(\tau;e_{\text{parse}});\mathcal{I}',\tau\hookrightarrow a\rangle;b} \grave{p} \rightsquigarrow p : \tau \dashv\Vdash^{\langle\mathcal{G};\Gamma\rangle}$ — by assumption
   (8) $\Delta \vdash p : \tau \dashv\Vdash \Gamma$ — by IH, part 2 on (7)

2. We induct on the premise. In the following, let $\hat{\Gamma} = \langle\mathcal{G};\Gamma\rangle$ and $\hat{\Phi} = \langle\mathcal{A};\Phi;\mathcal{I}\rangle$.

   Case (7.12a) through (7.12e). In each case, the proof is written identically to the proof of the corresponding case in the proof of Theorem B.28.

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:

$$\|\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\Vdash \hat{\Gamma}\| = \|\hat{p}\|$$
$$\|\vdash^{\Delta;\hat{\Phi};b} \grave{p} \rightsquigarrow p : \tau \dashv\Vdash^{\hat{\Gamma}}\| = \|b\|$$

where $\|b\|$ is the length of b and $\|\hat{p}\|$ is the sum of the lengths of the literal bodies in $\hat{p}$,

$$\begin{aligned}
\|\hat{x}\| &= 0 \\
\|\texttt{ufoldp}(\hat{p})\| &= \|\hat{p}\| \\
\|\texttt{utplp}[L](\{i \hookrightarrow \hat{p}_i\}_{i \in L})\| &= \sum_{i \in L} \|\hat{p}_i\| \\
\|\texttt{uinjp}[\ell](\hat{p})\| &= \|\hat{p}\| \\
\|\texttt{uapuptsm}[b][\hat{a}]\| &= \|b\| \\
\|\texttt{uplit}[b]\| &= \|b\|
\end{aligned}$$

The only case in the proof of part 1 that invokes part 2 are Case (7.7f) and (7.7g). There, we have that the metric remains stable:

$$\begin{aligned}
&\|\Delta \vdash_{\hat{\Phi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}})} \texttt{uapuptsm}[b][\hat{a}] \rightsquigarrow p : \tau \dashv\!| \, \hat{\Gamma}\| \\
=&\|\Delta \vdash_{\langle \mathcal{A}; \Phi', a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}}); \mathcal{I}', \tau \hookrightarrow a \rangle} \texttt{uplit}[b] \rightsquigarrow p : \tau \dashv\!| \, \hat{\Gamma}\| \\
=&\|\vdash^{\Delta; \hat{\Phi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}}); b} \hat{p} \rightsquigarrow p : \tau \dashv\!|^{\hat{\Gamma}}\| \\
=&\|b\|
\end{aligned}$$

The only case in the proof of part 2 that invokes part 1 is Case (7.12e). There, we have that $\mathsf{parseUPat}(\mathsf{subseq}(b; m; n)) = \hat{p}$ and the IH is applied to the judgement $\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!| \, \hat{\Gamma}$. 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:

$$\|\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!| \, \hat{\Gamma}\| < \|\vdash^{\Delta; \hat{\Phi}; b} \texttt{splicedp}[m; n] \rightsquigarrow p : \tau \dashv\!|^{\hat{\Gamma}}\|$$

i.e. by the definitions above,

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

This is established by appeal to Condition B.22, 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 delimit each literal body.

Condition 7.7 (Pattern Parsing Monotonicity). *If* $\mathsf{parseUPat}(b) = \hat{p}$ *then* $\|\hat{p}\| < \|b\|$.

Combining Conditions B.22 and 7.7, we have that $\|\hat{e}\| < \|b\|$ as needed. □

Finally, 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 ce-expression and ce-rule validation because the judgements are mutually defined.

Theorem 7.8 (Typed Expansion). *Letting* $\hat{\Psi} = \langle \mathcal{A}; \Psi; \mathcal{I} \rangle$, *if* $\Delta \vdash \Psi$ seTSMs *then all of the following hold:*

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

ii. *If* $\langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \rightsquigarrow r \Rightarrow \tau \mapsto \tau'$ *then* $\Delta\ \Gamma \vdash r : \tau \mapsto \tau'$.

(b) i. *If* $\langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau$ *and* $\Delta \vdash \tau$ type *then* $\Delta\ \Gamma \vdash e : \tau$.

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

ii. *If* $\Delta\ \Gamma \vdash^{\langle \mathcal{D};\Delta_{app} \rangle;\langle \mathcal{G};\Gamma_{app} \rangle;\hat{\Psi};\hat{\Phi};b} \grave{r} \rightsquigarrow r \Rightarrow \tau \mapsto \tau'$ *and* $\Delta \cap \Delta_{app} = \emptyset$ *and* $dom(\Gamma) \cap dom(\Gamma_{app}) = \emptyset$ *then* $\Delta \cup \Delta_{app}\ \Gamma \cup \Gamma_{app} \vdash r : \tau \mapsto \tau'$.

(b) i. *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* $\Delta \cap \Delta_{app} = \emptyset$ *and* $dom(\Gamma) \cap dom(\Gamma_{app}) = \emptyset$ *and* $\Delta \cup \Delta_{app} \vdash \tau$ type *then* $\Delta \cup \Delta_{app}\ \Gamma \cup \Gamma_{app} \vdash e : \tau$.

ii. *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* $\Delta \cap \Delta_{app} = \emptyset$ *and* $dom(\Gamma) \cap dom(\Gamma_{app}) = \emptyset$ *and* $\Delta \cup \Delta_{app} \vdash \tau'$ type *then* $\Delta \cup \Delta_{app}\ \Gamma \cup \Gamma_{app} \vdash r : \tau \mapsto \tau'$.

Proof. By mutual rule induction over Rules (7.1), Rules (7.3), Rule (7.5), Rule (7.6), Rules (7.8), Rules (7.9), Rule (7.10) and Rule (7.11). In the following, we refer to the induction hypothesis applied to the assumption $\Delta \vdash \Psi$ seTSMs as simply the "IH". When we apply the induction hypothesis to a different argument, we refer to it as the "Outer IH".

1. In the following, let $\hat{\Delta} = \langle \mathcal{D}; \Delta \rangle$ and $\hat{\Gamma} = \langle \mathcal{G}; \Gamma \rangle$. We have:

(a) i. We induct on the assumption.

Case (7.1a)**.** We have:

(1) $e = x$	by assumption
(2) $\Gamma = \Gamma', x : \tau$	by assumption
(3) $\Delta\ \Gamma', x : \tau \vdash x : \tau$	by Rule (**??**)

Case (7.1b)**.** We have:

(1) $\hat{e} = \texttt{uasc\{}\hat{\tau}\texttt{\}(}\hat{e}'\texttt{)}$	by assumption
(2) $\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau$ type	by assumption
(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \rightsquigarrow e \Leftarrow \tau$	by assumption
(4) $\Delta \vdash \tau$ type	by Lemma B.26 on (2)
(5) $\Delta\ \Gamma \vdash e : \tau$	by IH, part 1(b)(i) to (3) and (4)

Case (7.1c) through (7.1k)**.** In each of these cases, we apply:

- Lemma B.26 to or over all type expansion premises.
- The IH, part 1(a)(i) to or over all synthetic typed expression expansion premises.
- The IH, part 1(a)(ii) to or over all synthetic rule expansion premises.
- The IH, part 1(b)(i) to or over all analytic typed expression expansion premises.

We then derive the conclusion by applying Rules (B.2) and Rule (B.3) as needed.

Case (7.2a)**.** We have:

(1) $\hat{e} = \texttt{usyntaxue\{}e_{\text{parse}}\texttt{\}\{}\hat{\tau}'\texttt{\}(}\hat{a}.\hat{e}'\texttt{)}$	by assumption

(2) $\hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'$ type — by assumption
(3) $\emptyset\,\emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResultSE)}$ — by assumption
(4) $\hat{\Delta}\,\hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \texttt{setsm}(\tau'; e_{\text{parse}}); \mathcal{I} \rangle; \hat{\Phi}} \hat{e}' \rightsquigarrow e \Rightarrow \tau$ — by assumption
(5) $\Delta \vdash \Psi$ seTSMs — by assumption
(6) $\Delta \vdash \tau'$ type — by Lemma B.26 to (2)
(7) $\Delta \vdash \Psi, \hat{a} \hookrightarrow \texttt{setsm}(\tau'; e_{\text{parse}})$ seTSMs — by Definition **??** on (5), (6) and (3)
(8) $\Delta\,\Gamma \vdash e : \tau$ — by Outer IH, part 1(a)(i) on (7) and (4)

Case (7.2b)**.** We have:
(1) $\hat{e} = \texttt{uapuetsm}[b]\,[\hat{a}]$ — by assumption
(2) $\hat{\Psi} = \langle \mathcal{A}' \uplus \hat{a} \rightsquigarrow a; \Psi', a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}}); \mathcal{I} \rangle$ — by assumption
(3) $b \downarrow_{\text{Body}} e_{\text{body}}$ — by assumption
(4) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}[\texttt{Success}](e_{\text{proto}})$ — by assumption
(5) $e_{\text{proto}} \uparrow_{\text{ProtoExpr}} \grave{e}$ — by assumption
(6) $\emptyset\,\emptyset \vdash_{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e \Leftarrow \tau$ — by assumption
(7) $\Delta \vdash \Psi$ seTSMs — by assumption
(8) $\Delta \vdash \tau$ type — by Definition **??** on (7)
(9) $\emptyset \cap \Delta = \emptyset$ — by finite set intersection identity
(10) $\emptyset \cap \text{dom}(\Gamma) = \emptyset$ — by finite set intersection identity
(11) $\emptyset \cup \Delta\,\emptyset \cup \Gamma \vdash e : \tau$ — by IH, part 2(a)(i) on (6), (9), (10) and (8)
(12) $\Delta\,\Gamma \vdash e : \tau$ — by definition of finite set and finite function union over (11)

Case (7.2c)**.** We have:
(1) $\hat{e} = \texttt{uimplicite[}\hat{a}\texttt{](}\hat{e}\texttt{)}$ — by assumption
(2) $\hat{\Psi} = \langle \mathcal{A}' \uplus \hat{a} \rightsquigarrow a; \Psi', a \hookrightarrow \texttt{setsm}(\tau'; e_{\text{parse}}); \mathcal{I} \rangle$ — by assumption
(3) $\hat{\Delta}\,\hat{\Gamma} \vdash_{\langle \mathcal{A}' \uplus \hat{a} \rightsquigarrow a; \Psi', a \hookrightarrow \texttt{setsm}(\tau'; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle; \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau$ — by assumption
(4) $\Delta\,\Gamma \vdash e : \tau$ — by IH, part 1(a)(i) on (3)

Case (7.2d)**.** We have:
(1) $\hat{e} = \texttt{usyntaxup\{}e_{\text{parse}}\texttt{\}\{}\hat{\tau}'\texttt{\}(}\hat{a}.\hat{e}'\texttt{)}$ — by assumption
(2) $\hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'$ type — by assumption

(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi},\hat{a}\rightsquigarrow a\hookrightarrow \mathtt{sptsm}(\tau';e_{\mathrm{parse}})} \hat{e}' \rightsquigarrow e \Rightarrow \tau$ by assumption

(4) $\Delta\ \Gamma \vdash e : \tau$ by IH, part 1(a)(i) on (3)

Case (7.2e)**.** We have:

(1) $\hat{e} = \mathtt{uimplicitp[}\hat{a}\mathtt{](}\hat{e}\mathtt{)}$ by assumption

(2) $\hat{\Phi} = \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \mathtt{sptsm}(\tau'; e_{\mathrm{parse}}); \mathcal{I} \rangle$ by assumption

(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\langle \mathcal{A}\uplus\hat{a}\rightsquigarrow a;\Phi,a\hookrightarrow\mathtt{sptsm}(\tau';e_{\mathrm{parse}});\mathcal{I}\uplus\tau\hookrightarrow a\rangle} \hat{e} \rightsquigarrow e \Rightarrow \tau$ by assumption

(4) $\Delta\ \Gamma \vdash e : \tau$ by IH, part 1(a)(i) on (3)

ii. We induct on the assumption. There is one case.

Case (7.5)**.** We have:

(1) $\hat{r} = \mathtt{urule(}\hat{p}.\hat{e}\mathtt{)}$ by assumption

(2) $r = \mathtt{rule(}p.e\mathtt{)}$ by assumption

(3) $\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\Vert \langle \mathcal{A}'; \Gamma \rangle$ by assumption

(4) $\hat{\Delta}\ \langle \mathcal{A} \uplus \mathcal{A}'; \Gamma \cup \Gamma \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau'$ by assumption

(5) $\Delta \vdash p : \tau \dashv\Vert \Gamma$ by Theorem 7.6, part 1 on (3)

(6) $\Delta\ \Gamma \cup \Gamma \vdash e : \tau'$ by IH, part 1(a)(i) on (4)

(7) $\Delta\ \Gamma \vdash \mathtt{rule(}p.e\mathtt{)} : \tau \Mapsto \tau'$ by Rule (B.3) on (5) and (6)

(b) i. We induct on the assumption.

Case (7.3a)**.** We have:

(1) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau$ by assumption

(2) $\Delta\ \Gamma \vdash e : \tau$ by IH, part 1(a)(i) on (1)

Case (7.3b) through (7.3h)**.** In each of these cases, we apply:

- Lemma B.26 to or over all type expansion premises.
- The IH, part 1(a)(i) to or over all synthetic typed expression expansion premises.
- The IH, part 1(a)(ii) to or over all synthetic rule expansion premises.
- The IH, part 1(b)(i) to or over all analytic typed expression expansion premises.

We then derive the conclusion by applying Rules (B.2) and Rule (B.3) as needed.

Case (7.4a)**.** We have:

(3) $\hat{e} = \mathtt{usyntaxue\{}e_{\mathrm{parse}}\mathtt{\}\{}\hat{\tau}'\mathtt{\}(}\hat{a}.\hat{e}'\mathtt{)}$ by assumption

(4) $\hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'$ type by assumption

(5) , by assumption

(6) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{setsm}(\tau'; e_{\text{parse}}); \hat{\Phi}} \hat{e}' \rightsquigarrow e \Leftarrow \tau$ by assumption

(7) $\Delta \vdash \Psi$ seTSMs by assumption

(8) $\Delta \vdash \tau'$ type by Lemma B.26 to (4)

(9) $\Delta \vdash \Psi, \hat{a} \hookrightarrow \texttt{setsm}(\tau'; e_{\text{parse}})$ seTSMs by Definition **??** on (7), (8) and (5)

(10) $\Delta\ \Gamma \vdash e : \tau$ by IH, part 1(b)(i) on (6)

Case (7.4b)**.** We have:

(1) $\hat{e} = \texttt{uapuetsm}[b][\hat{a}]$ by assumption

(2) $\hat{\Psi} = \hat{\Psi}', \hat{a} \rightsquigarrow a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}})$ by assumption

(3) $b \downarrow_{\text{Body}} e_{\text{body}}$ by assumption

(4) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}[\texttt{Success}](e_{\text{proto}})$ by assumption

(5) $e_{\text{proto}} \uparrow_{\text{ProtoExpr}} \grave{e}$ by assumption

(6) $\varnothing\ \varnothing \vdash^{\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; \hat{\Phi}; b} \grave{e} \rightsquigarrow e \Leftarrow \tau$ by assumption

(7) $\varnothing \cap \Delta = \varnothing$ by finite set intersection identity

(8) $\varnothing \cap \text{dom}(\Gamma) = \varnothing$ by finite set intersection identity

(9) $\varnothing \cup \Delta\ \varnothing \cup \Gamma \vdash e : \tau$ by IH, part 2(b)(i) on (6), (7), and (8)

(10) $\Delta\ \Gamma \vdash e : \tau$ by definition of finite set union over (9)

Case (7.4c)**.** We have:

(1) $\hat{e} = \texttt{uelit}[b]$ by assumption

(2) $\hat{\Psi} = \langle \mathcal{A}; \Psi, a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle$ by assumption

(3) $b \downarrow_{\text{Body}} e_{\text{body}}$ by assumption

(4) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}[\texttt{Success}](e_{\text{proto}})$ by assumption

(5) $e_{\text{proto}} \uparrow_{\text{ProtoExpr}} \grave{e}$ by assumption

(6) $\varnothing\ \varnothing \vdash^{\hat{\Delta}; \hat{\Gamma}; \langle \mathcal{A}; \Psi, a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle; \hat{\Phi}; b} \grave{e} \rightsquigarrow e \Leftarrow \tau$ by assumption

(7) $\varnothing \cap \Delta = \varnothing$ by finite set intersection identity

(8) $\varnothing \cap \text{dom}(\Gamma) = \varnothing$ by finite set intersection identity

(9) $\varnothing \cup \Delta\ \varnothing \cup \Gamma \vdash e : \tau$ by IH, part 2(a)(i) on (6), (7), and (8)

(10) $\Delta\ \Gamma \vdash e : \tau$ by definition of finite set union over (9)

Case (7.4d)**.** We have:

(1) $\hat{e} = \texttt{usyntaxup}\{e_{\text{parse}}\}\{\hat{\tau}'\}(\hat{a}.\hat{e}')$ by assumption
(2) $\hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'$ type by assumption
(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{sptsm}(\tau';e_{\text{parse}})} \hat{e}' \rightsquigarrow e \Leftarrow \tau$ by assumption
(4) $\Delta\ \Gamma \vdash e : \tau$ by IH, part 1(b)(i) on (3)

Case (7.4e). We have:
(1) $\hat{e} = \texttt{uimplicitp}[\hat{a}](\hat{e})$ by assumption
(2) $\hat{\Phi} = \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{sptsm}(\tau'; e_{\text{parse}}); \mathcal{I} \rangle$ by assumption
(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\langle \mathcal{A}\uplus\hat{a}\rightsquigarrow a;\Phi,a\hookrightarrow\texttt{sptsm}(\tau';e_{\text{parse}});\mathcal{I}\uplus\tau\hookrightarrow a\rangle} \hat{e} \rightsquigarrow e \Leftarrow \tau$ by assumption
(4) $\Delta\ \Gamma \vdash e : \tau$ by IH, part 1(b)(i) on (3)

ii. We induct on the assumption. There is one case.

Case (7.6). We have:
(1) $\hat{r} = \texttt{urule}(\hat{p}.\hat{e})$ by assumption
(2) $r = \texttt{rule}(p.e)$ by assumption
(3) $\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\Vdash \langle \mathcal{A}'; \Gamma \rangle$ by assumption
(4) $\hat{\Delta}\ \langle \mathcal{A} \uplus \mathcal{A}'; \Gamma \cup \Gamma \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'$ by assumption
(5) $\Delta \vdash p : \tau \dashv\Vdash \Gamma$ by Theorem 7.6, part 1 on (3)
(6) $\Delta\ \Gamma \cup \Gamma \vdash e : \tau'$ by IH, part 1(b)(i) 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 $\hat{\Gamma} = \langle \mathcal{G}; \Gamma_{\text{app}} \rangle$ and $\mathbb{E} = \hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; \hat{\Phi}; b$.

(a) i. We induct on the assumption.

Case (7.8a). We have:
(1) $e = x$ by assumption
(2) $\Gamma = \Gamma', x : \tau$ by assumption
(3) $\Delta\ \Gamma', x : \tau \vdash x : \tau$ by Rule (**??**)

Case (7.8b). We have:
(1) $\grave{e} = \texttt{prasc}\{\grave{\tau}\}(\grave{e}')$ by assumption
(2) $\Delta \cap \Delta_{\text{app}} = \emptyset$ by assumption
(3) $\text{dom}(\Gamma) \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset$ by assumption
(4) $\Delta \vdash^{\text{ts}(\mathbb{E})} \grave{\tau} \rightsquigarrow \tau$ type by assumption
(5) $\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e}' \rightsquigarrow e \Leftarrow \tau$ by assumption
(6) $\Delta \cup \Delta_{\text{app}} \vdash \tau$ type by Lemma B.27 on (4)
(7) $\Delta\ \Gamma \vdash e : \tau$ by IH, part 2(b)(i) to (5), (2), (3) and (6)

Case (7.8c) through (7.8k)**.** In each of these cases, we apply:

- Lemma B.27 to or over all ce-type validation premises.
- The IH, part 2(a)(i) to or over all synthetic ce-expression validation premises.
- The IH, part 2(a)(ii) to or over all synthetic ce-rule validation premises.
- The IH, part 2(b)(i) to or over all analytic ce-expression validation premises.

We then derive the conclusion by applying Rules (B.2), Rule (B.3), Lemma B.2, the identification convention and exchange as needed.

Case (7.8l)**.** We have:

(1)	$\grave{e} = \texttt{splicede}[m; n]$	by assumption
(2)	$\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e}$	by assumption
(3)	$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau$	by assumption
(4)	$\Delta \cap \Delta_{\text{app}} = \emptyset$	by assumption
(5)	$\text{dom}(\Gamma) \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset$	by assumption
(6)	$\Delta_{\text{app}}\ \Gamma_{\text{app}} \vdash e : \tau$	by IH, part 1(a)(i) on (3)
(7)	$\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash e : \tau$	by Lemma B.2 over Δ and Γ and exchange on (6)

ii. We induct on the assumption. There is one case.

Case (7.10)**.** We have:

(1)	$\grave{r} = \texttt{prrule}(p.\grave{e})$	by assumption
(2)	$r = \texttt{rule}(p.e)$	by assumption
(3)	$\Delta \vdash p : \tau \dashv \Gamma$	by assumption
(4)	$\Delta\ \Gamma \cup \Gamma \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e \Rightarrow \tau'$	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)(i) 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)	$\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash \texttt{rule}(p.e) : \tau \Rrightarrow \tau'$	by Rule (B.3) on (3) and (11)

(b) i. We induct on the assumption.

Case (7.9a)**.** We have:

(1)	$\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau$	by assumption
(2)	$\Delta\ \Gamma \vdash e : \tau$	by IH, part 2(a)(i) on (1)

Case (7.9b) through (7.3h)**.** In each of these cases, we apply:

- Lemma B.27 to or over all ce-type validation premises.
- The IH, part 2(a)(i) to or over all synthetic ce-expression validation premises.
- The IH, part 2(a)(ii) to or over all synthetic ce-rule validation premises.
- The IH, part 2(b)(i) to or over all analytic ce-expression validation premises.

We then derive the conclusion by applying Rules (B.2), Rule (B.3), Lemma B.2, the identification convention and exchange as needed.

Case (7.9i)**.** We have:

(3)	$\grave{e} = \texttt{splicede}[m;n]$	by assumption
(4)	$\mathsf{parseUExp}(\mathsf{subseq}(b;m;n)) = \hat{e}$	by assumption
(5)	$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau$	by assumption
(6)	$\Delta \cup \Delta_{\mathrm{app}} \vdash \tau\ \mathsf{type}$	by assumption
(7)	$\Delta \cap \Delta_{\mathrm{app}} = \emptyset$	by assumption
(8)	$\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma_{\mathrm{app}}) = \emptyset$	by assumption
(9)	$\Delta_{\mathrm{app}}\ \Gamma_{\mathrm{app}} \vdash e : \tau$	by IH, part 1(b)(i) on (5), (7), (8) and (6)
(10)	$\Delta \cup \Delta_{\mathrm{app}}\ \Gamma \cup \Gamma_{\mathrm{app}} \vdash e : \tau$	by Lemma B.2 over Δ and Γ and exchange on (9)

ii. We induct on the assumption. There is one case.

Case (7.11)**.** We have:

(1)	$\grave{r} = \texttt{prrule(}p.\grave{e}\texttt{)}$	by assumption
(2)	$r = \texttt{rule(}p.e\texttt{)}$	by assumption
(3)	$\Delta \vdash p : \tau \dashv\Vert\ \Gamma$	by assumption
(4)	$\Delta\ \Gamma \cup \Gamma \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e \Leftarrow \tau'$	by assumption
(5)	$\Delta \cup \Delta_{\mathrm{app}} \vdash \tau'\ \mathsf{type}$	by assumption
(6)	$\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma_{\mathrm{app}}) = \emptyset$	by assumption
(7)	$\Delta \cap \Delta_{\mathrm{app}} = \emptyset$	by assumption
(8)	$\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma) = \emptyset$	by identification convention
(9)	$\mathrm{dom}(\Gamma_{\mathrm{app}}) \cap \mathrm{dom}(\Gamma) = \emptyset$	by identification convention

(10)	$\mathrm{dom}(\Gamma \cup \Gamma) \cap \mathrm{dom}(\Gamma_{\mathrm{app}}) = \emptyset$	by standard finite set definitions and identities on (8), (9) and (6)
(11)	$\Delta \cup \Delta_{\mathrm{app}}\ \Gamma \cup \Gamma \cup \Gamma_{\mathrm{app}} \vdash e : \tau'$	by IH, part 2(b)(i) on (4), (7), (10) and (5)
(12)	$\Delta \cup \Delta_{\mathrm{app}}\ \Gamma \cup \Gamma_{\mathrm{app}} \cup \Gamma \vdash e : \tau'$	by exchange of Γ and Γ_{app} on (11)
(13)	$\Delta \cup \Delta_{\mathrm{app}}\ \Gamma \cup \Gamma_{\mathrm{app}} \vdash \mathtt{rule}(p.e) : \tau \Rightarrowtriangle \tau'$	by Rule (B.3) on (3) and (12)

We must now show that the induction is well-founded. All applications of the IH are on subterms except the following.

- The only cases in the proof of part 1 that invoke the IH, part 2 are Case (7.2b) in the proof of part 1(a)(i) and Case (7.4c) in the proof of part 1(b)(i). The only cases in the proof of part 2 that invoke the IH, part 1 are Case (7.8l) in the proof of part 2(a)(i) and Case (7.9i) in the proof of part 2(b)(i). We can show that the following metric on the judgements that we induct on is stable in one direction and strictly decreasing in the other direction:

$$\begin{aligned}
\|\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau\| &= \|\hat{e}\| \\
\|\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau\| &= \|\hat{e}\| \\
\|\Delta\ \Gamma \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e \Rightarrow \tau\| &= \|b\| \\
\|\Delta\ \Gamma \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e \Leftarrow \tau\| &= \|b\|
\end{aligned}$$

  where $\|b\|$ is the length of b and $\|\hat{e}\|$ is the sum of the lengths of the ueTSM literal bodies in $\hat{e}$,

$$\begin{aligned}
\|\hat{x}\| &= 0 \\
\|\mathtt{uasc\{}\hat{\tau}\mathtt{\}(}\hat{e}\mathtt{)}\| &= \|\hat{e}\| \\
\|\mathtt{uletval(}\hat{e}\mathtt{;}\,\hat{x}.\hat{e}'\mathtt{)}\| &= \|\hat{e}\| + \|\hat{e}'\| \\
\|\mathtt{uanalam(}\hat{x}.\hat{e}\mathtt{)}\| &= \|\hat{e}\| \\
\|\mathtt{ulam\{}\hat{\tau}\mathtt{\}(}\hat{x}.\hat{e}\mathtt{)}\| &= \|\hat{e}\| \\
\|\mathtt{uap(}\hat{e}_1\mathtt{;}\,\hat{e}_2\mathtt{)}\| &= \|\hat{e}_1\| + \|\hat{e}_2\| \\
\|\mathtt{utlam(}\hat{t}.\hat{e}\mathtt{)}\| &= \|\hat{e}\| \\
\|\mathtt{utap\{}\hat{\tau}\mathtt{\}(}\hat{e}\mathtt{)}\| &= \|\hat{e}\| \\
\|\mathtt{ufold(}\hat{e}\mathtt{)}\| &= \|\hat{e}\| \\
\|\mathtt{uunfold(}\hat{e}\mathtt{)}\| &= \|\hat{e}\| \\
\|\mathtt{utpl\{}L\mathtt{\}(\{}i \hookrightarrow \hat{e}_i\mathtt{\}}_{i \in L}\mathtt{)}\| &= \sum_{i \in L} \|\hat{e}_i\| \\
\|\mathtt{uprj[}\ell\mathtt{](}\hat{e}\mathtt{)}\| &= \|\hat{e}\|
\end{aligned}$$

$$
\begin{aligned}
\|\texttt{uin[}\ell\texttt{](}\hat{e}\texttt{)}\| &= \|\hat{e}\| \\
\|\texttt{umatch[}n\texttt{](}\hat{e}\texttt{;}\{\hat{r}_i\}_{1\leq i\leq n}\texttt{)}\| &= \|\hat{e}\| + \sum_{1\leq i\leq n} \|r_i\| \\
\|\texttt{usyntaxue\{}e_{\text{parse}}\texttt{\}\{}\hat{\tau}\texttt{\}(}\hat{a}\texttt{.}\hat{e}\texttt{)}\| &= \|\hat{e}\| \\
\|\texttt{uimplicite[}\hat{a}\texttt{](}\hat{e}\texttt{)}\| &= \|\hat{e}\| \\
\|\texttt{uapuetsm[}b\texttt{][}\hat{a}\texttt{]}\| &= \|b\| \\
\|\texttt{uelit[}b\texttt{]}\| &= \|b\| \\
\|\texttt{usyntaxup\{}e_{\text{parse}}\texttt{\}\{}\hat{\tau}\texttt{\}(}\hat{a}\texttt{.}\hat{e}\texttt{)}\| &= \|\hat{e}\| \\
\|\texttt{uimplicitp[}\hat{a}\texttt{](}\hat{e}\texttt{)}\| &= \|\hat{e}\|
\end{aligned}
$$

and $\|r\|$ is defined as follows:

$$\|\texttt{urule(}\hat{p}\texttt{.}\hat{e}\texttt{)}\| = \|\hat{e}\|$$

Going from part 1 to part 2, the metric remains stable:

$$
\begin{aligned}
&\|\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{uapuetsm[}b\texttt{][}\hat{a}\texttt{]} \rightsquigarrow e \Rightarrow \tau\| \\
=&\|\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{uelit[}b\texttt{]} \rightsquigarrow e \Leftarrow \tau\| \\
=&\|\varnothing\ \varnothing \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e \Leftarrow \tau\| \\
=&\|b\|
\end{aligned}
$$

Going from part 2 to part 1, in each case we have that $\mathsf{parseUExp}(\mathsf{subseq}(b;m;n)) = \hat{e}$ and the IH is applied to the judgements $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau$ and $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau$, respectively. 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:

$$\|\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau\| < \|\Delta\ \Gamma \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \texttt{splicede[}m;n\texttt{]} \rightsquigarrow e \Rightarrow \tau\|$$

and

$$\|\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau\| < \|\Delta\ \Gamma \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \texttt{splicede[}m;n\texttt{]} \rightsquigarrow e \Leftarrow \tau\|$$

i.e. by the definitions above,

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

This is established by appeal to Condition B.22, which states that subsequences of b are no longer than b, and the following condition, 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 delimit each literal body.

Condition 7.9 (Expression Parsing Monotonicity)**.** *If* $\mathsf{parseUExp}(b) = \hat{e}$ *then* $\|\hat{e}\| < \|b\|$.

Combining Conditions B.22 and 7.9, we have that $\|\hat{e}\| < \|b\|$ as needed.

- In Case (7.3a) of the proof of part 1(b)(i), we apply the IH, part 1(a)(i), with $\hat{e} = \hat{e}$. This is well-founded because all applications of the IH, part 1(b)(i) elsewhere in the proof are on strictly smaller terms.
- Similarly, in Case (7.9a) of the proof of part 2(b)(i), we apply the IH, part 2(a)(i), with $\grave{e} = \grave{e}$. This is well-founded because all applications of the IH, part 2(b)(i) elsewhere in the proof are on strictly smaller terms.

□

7.3 Related Work

TODO: cite/comment on past work on bidirectional typechecking TODO: TSLs TODO: ichikawa modularity paper TODO: other things from related work section of ECOOP paper

Part IV

Conclusion

Chapter 8

Discussion & Future Directions

> *Science is an endless search for truth. Any representation of reality we develop can be only partial. There is no finality, sometimes no single best representation. There is only deeper understanding, more revealing and enveloping representations.*
>
> – Carl R. Woese [**?**]

8.1 Interesting Applications

Most of the examples in Sec. 2.3 can be expressed straightforwardly using the constructs introduced in the previous chapters. Here, let us highlight certain interesting examples and exceptions.

8.1.1 Monadic Commands

8.2 Summary

TODO: Write summary

8.3 Future Directions

We did not consider situations where a library clients wants to provide derived forms for defining types, signatures, modules or other declarations (though we have explored syntax for types in a recent short paper [78].) We also did not consider situations where a library client wants to generate an expression or a pattern based on the structure of a type (e.g. automatic generation of equality comparisons.) Finally, we do not consider situations that require modifications to the underlying type structure of a language (though "reasonably programmable type structure" is a rich avenue for future work.)

8.3.1 TSM Packaging

In the exposition thusfar, we have assumed that TSMs have delimited scope. However, ideally, we would like to be able to define TSMs within a module:

```
structure Rxlib = struct
  type Rx = (* ... *)
  syntax $rx at Rx { (* ... *) }
  (* ... *)
end
```

However, this leads to an important question: how can we write down a signature for the module `Rxlib`? One approach would be to simply duplicate the full definition of the TSM in the signature, but this leads to inelegant code duplication and raises the difficult question of how the language should decide whether one TSM is a duplicate of another. For this reason, in VerseML, a signature can only refer to a previously defined TSM. So, for example, we can write down a signature for `Rxlib` after it has been defined:

```
signature RXLIB = sig
  type Rx = (* ... *)
  syntax $rx = Rxlib.$rx
  (* ... *)
end
Rxlib : RXLIB (* check Rxlib against RXLIB after the fact *)
```

Alternatively, we can define the type `Rx` and the TSM `$rx` before defining `Rxlib`:

```
local
  type Rx = (* ... *)
  syntax $rx at Rx { (* ... *) }
in
  structure Rxlib :
  sig
    type Rx = Rx
    syntax $rx = $rx
    (* ... *)
  end = struct
    type Rx = Rx
    syntax $rx = $rx
    (* ... *)
  end
end
```

Another important question is: how does a TSM defined within a module at a type that is held abstract outside of that module operate? For example, consider the following:

```
local
  type Rx = (* ... *)
  syntax $rx at Rx { (* ... *) }
in
  structure Rxlib :
  sig
    type Rx (* held abstract *)
```

```
    syntax $rx = $rx
    (* ... *)
  end = struct
    type Rx = Rx
    syntax $rx = $rx
    (* ... *)
  end
end
```

If we apply `Rxlib.$rx`, it may generate an expansion that uses the constructors of the `Rx` type. However, because the type is being held abstract, these constructors may not be visible at the application site. TODO: actually, this is why doing this is a bad idea. export TSMs only from units, not modules

8.3.2 TSLs

8.4 pTSLs By Example

For example, a module `P` can associate the TSM `rx` defined in the previous section with the abstract type `R.t` by qualifying the definition of the sealed module it is defined by as follows:

```
module R = mod {
  type t = (* ... *)
  (* ... *)
} :> RX with syntax rx
```

More generally, when sealing a module expression against a signature, the programmer can specify, for each abstract type that is generated, at most one previously defined TSMs. This TSM must take as its first parameter the module being sealed.

The following function has the same expansion as `example_using_tsm` but, by using the TSL just defined, it is more concise. Notice the return type annotation, which is necessary to ensure that the TSL can be unambiguously determined:

```
fun example_using_tsl(name : string) : R.t => /@name: %ssn/
```

As another example, let us consider the standard list datatype. We can use TSLs to express derived list syntax, for both expressions and patterns:

```
datatype list('a) { Nil | Cons of 'a * list('a) } with syntax {
  static fn (body : Body) =>
    (* ... comma-delimited spliced exps ... *)
} with pattern syntax {
  static fn (body : Body) : Pat =>
    (* ... list pattern parser ... *)
}
```

Together with the TSL for regular expression patterns, this allows us to write lists like this:

```
let val x : list(R.t) = [/\d/, /\d\d/, /\d\d\d/]
```

From the client's perspective, it is essentially as if the language had built in derived syntax for lists and regular expression patterns directly.

8.5 Parameterized Modules

TSLs can be associated with abstract types that are generated by parameterized modules (i.e. generative functors in Standard ML) as well. For example, consider a trivially parameterized module that creates modules sealed against `RX`:

```
module F() => mod {
  type t = (* ... *)
  (* ... *)
} :> RX with syntax rx
```

Each application of `F` generates a distinct abstract type. The semantics associates the appropriately parameterized TSM with each of these as they are generated:

```
module F1 = F() (* F1.t has TSL rx(F1) *)
module F2 = F() (* F2.t has TSL rx(F2) *)
```

As a more complex example, let us define two signatures, `A` and `B`, a TSM `$G` and a parameterized module `G : A -> B`:

```
signature A = sig { type t; val x : t }
signature B = sig { type u; val y : u }
syntax $G(M : A)(G : B) at G.u { (* ... *) }
module G(M : A) => mod {
  type u = M.t; val y = M.x } :> B with syntax $G(M)
```

Both `G` and `$G` take a parameter `M : A`. We associate the partially applied TSM `$G(M)` with the abstract type that `G` generates. Again, this satisfies the requirement that one must be able to apply the TSM being associated with the abstract type to the module being sealed.

Only fully abstract types can have TSLs associated with them. Within the definition of `G`, type `u` does not have a TSL available to it because it is synonymous to `M.t`. More generally, TSL lookup respects type equality, so any synonyms of a type with a TSL will also have that TSL. We can see this in the following example, where the type `u` has a different TSL associated with it inside and outside the definition of the module `N`:

```
module M : A = mod { type t = int; val x = 0 }
module G1 = G(M) (* G1.t has TSL $G(M), per above *)
module N = mod {
  type u = G1.t (* u = G1.t in this scope, so u also has TSL $G(M) *)
  val y = /asdf/ (* we can use it to create a value of that type *)
} :> B (* did not specify a TSL for N.u at the point where it is sealed,
          so N.u has no TSL in the outer scope *)
val z : N.u = /asdf/ (* ERROR: no TSL for type N.u *)
```

8.6 miniVerse$_{\text{TSL}}$

A formal specification of TSLs in a language that supports only non-parametric datatypes is available in a paper published in ECOOP 2014 [77].

8.6.1 TSMs and TSLs In Candidate Expansions

Candidate expansions cannot themselves define or apply TSMs. This simplifies our metatheory, though it can be inconvenient at times for TSM providers. We discuss adding the ability to use TSMs within candidate expansions here.TODO: write this

8.6.2 Pattern Matching Over Values of Abstract Type

ML does not presently support pattern matching over values of an abstract data type. However, there have been proposals for adding support for pattern matching over abstract data types defined by modules having a "datatype-like" shape, e.g. those that define a case analysis function like the one specified by `RX`,

8.6.3 Integration Into Other Languages

We conjecture that the constructs we describe could be integrated into dependently typed functional languages, e.g. Coq, but leave the technical developments necessary for doing so as future work.

Some of the constructs in Chapter 3, Chapter 5 and Chapter 7 could also be adapted for use in imperative languages with non-trivial type structure, like Java. Similarly, some of the constructs we discuss could also be adapted into "dynamic languages" like Racket or Python, though the constructs in Chapter 7 are not relevant to such languages.

8.6.4 Mechanically Verifying TSM Definitions

Finally, VerseML is not designed for advanced theorem proving tasks where languages like Coq, Agda or Idris might be used today. That said, we conjecture that the primitives we describe could be integrated into languages like Gallina (the "external language" of the Coq proof assistant [69]) with modifications, but do not plan to pursue this line of research here.

In such a setting, you could verify TSM definitions TODO: finish writing this

8.6.5 Improved Error Reporting

8.6.6 Controlled Binding

8.6.7 Type-Aware Splicing

8.6.8 Integration With Code Editors

8.6.9 Resugaring

TODO: Cite recent work at PLDI (?) and ICFP from Brown

8.6.10 Non-Textual Display Forms

TODO: Talk about active code completion work and future ideas

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 to the author by email at the following address:

comar@cs.cmu.edu

The author will also review and accept pull requests on GitHub.

Bibliography

TODO (Later): List conference abbreviations.
TODO (Later): Remove extraneous nonsense from entries.

[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] Ocaml batteries included. http://batteries.forge.ocamlcore.org/. Retrieved May 31, 2016., . 1.1.2

[3] core - jane street capital's standard library overlay. https://github.com/janestreet/core/. Retrieved May 31, 2016., . 1.1.2

[4] [Rust] Macros. https://doc.rust-lang.org/book/macros.html. Retrieved Nov. 3, 2015. 3.1.2

[5] SML/NJ Quote/Antiquote. http://www.smlnj.org/doc/quote.html, . 2.4.2

[6] The Visible Compiler. http://www.smlnj.org/doc/Compiler/pages/compiler.html, . 3.1.2

[7] Information technology portable operating system interface (posix) base specifications, issue 7. *Information technology Portable Operating System Interface (POSIX) Base Specifications, Issue 7*. 2.4.1

[8] OWASP Top 10 2013. https://www.owasp.org/index.php/Top_10_2013-Top_10, 2013. 3

[9] Annika Aasa. Precedences in specifications and implementations of programming languages. *Theor. Comput. Sci.*, 142(1):3–26, 1995. doi: 10.1016/0304-3975(95)90680-J. URL http://dx.doi.org/10.1016/0304-3975(95)90680-J. 2.4.4

[10] Annika Aasa, Kent Petersson, and Dan Synek. Concrete syntax for data objects in functional languages. In *LISP and Functional Programming*, pages 96–105, 1988. doi: 10.1145/62678.62688. URL http://doi.acm.org/10.1145/62678.62688. 2.4.8

[11] Peter Aczel. A general Church-Rosser theorem. Technical report, Technical Report, University of Manchester, 1978. 1.1

[12] Michael D. Adams. Towards the Essence of Hygiene. In *POPL*, 2015. doi: 10.1145/2676726.2677013. URL http://doi.acm.org/10.1145/2676726.2677013. 2.4.10

[13] Andrew W. Appel and David B. MacQueen. Standard ML of new jersey. In *PLILP*, pages 1–13, 1991. doi: 10.1007/3-540-54444-5_83. URL http://dx.doi.org/10.1007/3-540-54444-5_83. 2.1

[14] George EP Box and Norman R Draper. *Empirical model-building and response surfaces*. John Wiley & Sons, 1987. 2.2

[15] 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*, 2005. 2.4.8

[16] Martin Bravenboer, Eelco Dolstra, and Eelco Visser. Preventing Injection Attacks with Syntax Embeddings. In *GPCE '07*, pages 3–12, 2007. ISBN 978-1-59593-855-8. doi: 10.1145/1289971.1289975. URL http://doi.acm.org/10.1145/1289971.1289975. 3

[17] 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, 2.4.10

[18] 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

[19] Adam Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 122–133. ACM, 2010. ISBN 978-1-4503-0019-3. URL http://doi.acm.org/10.1145/1806596.1806612. 1.2.1

[20] Adam Chlipala. Ur/Web: A simple model for programming the web. In *POPL '15*, pages 153–165, 2015. ISBN 978-1-4503-3300-9. URL http://dl.acm.org/citation.cfm?id=2676726. 1.2.1

[21] William D. Clinger and Jonathan Rees. Macros that work. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*, pages 155–162, 1991. doi: 10.1145/99583.99607. URL http://doi.acm.org/10.1145/99583.99607. 2.4.10

[22] Russ Cox, Tom Bergan, Austin T. Clements, M. Frans Kaashoek, and Eddie Kohler. Xoc, an extension-oriented compiler for systems programming. In *ASPLOS*, 2008. ISBN 978-1-59593-958-6. URL http://doi.acm.org/10.1145/1346281.1346312. 2.4.9

[23] Nils Anders Danielsson and Ulf Norell. Parsing mixfix operators. In *Implementation and Application of Functional Languages - 20th International Symposium, IFL 2008, Hatfield, UK, September 10-12, 2008. Revised Selected Papers*, pages 80–99, 2008. doi: 10.1007/978-3-642-24452-0_5. URL http://dx.doi.org/10.1007/978-3-642-24452-0_5. 2.4.4, 2.4.4

[24] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 184–195, 1996. doi: 10.1109/LICS.1996.561317. URL http://dx.doi.org/10.1109/LICS.1996.561317. 3

[25] 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

[26] Lukas Diekmann and Laurence Tratt. Eco: A language composition editor. In *International Conference on Software Language Engineering*, pages 82–101. Springer International Publishing, 2014. 2.4.8

[27] Derek Dreyer. *Understanding and evolving the ml module system*. PhD thesis, Citeseer, 2005. 5.2.1

[28] Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. Modular type classes. In Martin Hofmann 0001 and Matthias Felleisen, editors, *POPL*, pages 63–70. ACM, 2007. ISBN 1-59593-575-4. URL http://dl.acm.org/citation.cfm?id=1190216. 1.4

[29] Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*, 2013. ISBN 978-1-4503-2326-0. URL http://dl.acm.org/citation.cfm?id=2500365. 1.3.1

[30] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1992. 2.4.10

[31] Sebastian Erdweg and Felix Rieger. A framework for extensible languages. In *GPCE*, 2013. 1.2.1, 2.4.5

[32] Sebastian Erdweg, Tillmann Rendel, Christian Kastner, and Klaus Ostermann. SugarJ: Library-based syntactic language extensibility. In *OOPSLA '11*, pages 187–188, 2011. 1.2.1, 2.4.5

[33] Sebastian Erdweg, Lennart C. L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser. Growing a language environment with editor libraries. *ACM SIGPLAN Notices*, 47(3):167–176, March 2012. ISSN 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). doi: http://dx.doi.org/10.1145/2189751.2047891. 2

[34] 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, 3

[35] Sebastian Erdweg, Tijs van der Storm, and Yi Dai. Capture-avoiding and hygienic program transformations. In Richard Jones, editor, *ECOOP*, volume 8586 of *Lecture Notes in Computer Science*, pages 489–514. Springer, 2014. ISBN 978-3-662-44201-2. URL http://dx.doi.org/10.1007/978-3-662-44202-9. 3

[36] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009. 2.4.5

[37] Matthew Flatt. Creating languages in Racket. *Commun. ACM*, 55(1):48–56, January 2012. ISSN 0001-0782. doi: 10.1145/2063176.2063195. URL http://doi.acm.org/10.1145/2063176.2063195. 1.2.1, 2.4.5

[38] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT sympo-*

sium on Principles of programming languages, pages 111–122, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X. doi: 10.1145/964001.964011. URL http://pdos.csail.mit.edu/~baford/packrat/popl04/peg-popl04.pdf. 2.4.5, 2.4.8

[39] Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time. *CoRR*, abs/cs/0603077, 2006. URL http://arxiv.org/abs/cs/0603077. 2.4.5

[40] Steven Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *ICFP '01*, pages 74–85, 2001. 3

[41] 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

[42] 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

[43] T.G. Griffin. Notational definition-a formal account. In *Logic in Computer Science (LICS '88)*, pages 372–383, 1988. doi: 10.1109/LICS.1988.5134. 2.4.4

[44] 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. ISSN 0164-0925. doi: 10.1145/227699.227700. URL http://doi.acm.org/10.1145/227699.227700. 1.4

[45] 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

[46] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012. 4.2, 4.2.3

[47] 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, 5.2.1, 5.2.2, A.1

[48] Robert Harper, John C Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 341–354. ACM, 1989. 5.2.2

[49] 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

[50] 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

[51] David Herman. *A Theory of Typed Hygienic Macros*. PhD thesis, Northeastern University, Boston, MA, May 2010. 2.4.10

[52] David Herman and Mitchell Wand. A theory of hygienic macros. In *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008,*

Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, pages 48–62, 2008. doi: 10.1007/978-3-540-78739-6_4. URL http://dx.doi.org/10.1007/978-3-540-78739-6_4. 2.4.10

[53] John E Hopcroft. *Introduction to automata theory, languages, and computation*. Pearson Education India, 1979. 2.4.5

[54] 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

[55] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *Journal of functional programming*, 8(04):437–444, 1998. 2.4.6

[56] Peter Zilahy Ingerman. “pĀnini-backus form” suggested. *Commun. ACM*, 10(3):137–, March 1967. ISSN 0001-0782. doi: 10.1145/363162.363165. URL http://doi.acm.org/10.1145/363162.363165. 2.4.5

[57] 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. ISBN number 0-13-020249-5 (pbk). 3

[58] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, volume 1, pages 203–233, 2001. 2.4.9

[59] Antti-Juhani Kaijanaho. Evidence-based programming language design: a philosophical and methodological exploration. 2015. 2.2

[60] Lennart CL Kats and Eelco Visser. *The spoofax language workbench: rules for declarative specification of languages and IDEs*, volume 45. ACM, 2010. 2.4.5, 2

[61] 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

[62] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of standard ML. In Martin Hofmann 0001 and Matthias Felleisen, editors, *POPL*, pages 173–184. ACM, 2007. ISBN 1-59593-575-4. URL http://dl.acm.org/citation.cfm?id=1190216. 5.2.1

[63] 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. 1.2.1, 2, 2.4.5, 2.4.9, 4.1.4

[64] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *ACM Sigplan Notices*, volume 9, pages 50–59. ACM, 1974. 2.3.2

[65] Florian Lorenzen and Sebastian Erdweg. Modular and automated type-soundness verification for language extensions. In Greg Morrisett and Tarmo Uustalu, editors, *ICFP*, pages 331–342. ACM, 2013. ISBN 978-1-4503-2326-0. URL http://dl.acm.

org/citation.cfm?id=2500365. 2.4.5

[66] Florian Lorenzen and Sebastian Erdweg. Sound type-dependent syntactic language extension. In Rastislav Bodík and Rupak Majumdar, editors, *POPL*, pages 204–216. ACM, 2016. ISBN 978-1-4503-3549-2. URL http://dl.acm.org/citation.cfm?id=2837614. 2.4.5, 5

[67] David MacQueen. Modules for Standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 198–207, 1984. ISBN 0-89791-142-3. doi: 10.1145/800055.802036. URL http://doi.acm.org/10.1145/800055.802036. 2.3.2

[68] Michel Mauny and Daniel de Rauglaudre. A complete and realistic implementation of quotations for ml. In *Proc. 1994 Workshop on ML and its applications*, pages 70–78. Citeseer, 1994. 2.4.10

[69] The 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, 3, 8.6.4

[70] J. McCarthy. History of lisp. In *History of programming languages I*, pages 173–185. ACM, 1978. 3

[71] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997. 1.1.2

[72] Stephan Albert Missura. *Higher-Order Mix x Syntax for Representing Mathematical Notation and its Parsing*. PhD thesis, ETH) ZURICH, 1997. 2.4.4

[73] 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

[74] Catarina Dutilh Novaes. *Formal languages in logic: A philosophical and cognitive analysis*. Cambridge University Press, 2012. 2.2

[75] 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

[76] Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers. Active Code Completion. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE '12)*, pages 859–869, 2012. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337324. 2.4.1, 1

[77] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely composable type-specific languages. In *ECOOP*, 2014. 3.1, 3.2.5, 8.6

[78] Cyrus Omar, Chenglong Wang, and Jonathan Aldrich. Composable and hygienic typed syntax macros. In *ACM Symposium on Applied Computing (SAC)*, 2015. 3.1, 8.3

[79] J.F. Pane and B.A. Myers. *Usability issues in the design of novice programming systems*.

Citeseer, 1996. 1.2.2, 2.2

[80] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. 2.1, 3.2.2

[81] 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

[82] J C Reynolds. Types, abstraction and parametric polymorphism. In *IFIP'83, Paris, France*. North-Holland, September 1983. 5

[83] Nico Ritschel and Sebastian Erdweg. Modular capture avoidance for program transformations. In Richard F. Paige, Davide Di Ruscio, and Markus Völter, editors, *SLE*, pages 59–70. ACM, 2015. ISBN 978-1-4503-3686-4. URL http://dl.acm.org/citation.cfm?id=2814251. 3, 5

[84] August Schwerdfeger and Eric Van Wyk. Verifiable composition of deterministic grammars. In *PLDI '09*, pages 199–210, 2009. ISBN 978-1-60558-392-1. URL http://doi.acm.org/10.1145/1542476.1542499. 1.2.1, 2.4.8, 1

[85] Denys Shabalin, Eugene Burmako, and Martin Odersky. Quasiquotes for Scala. Technical Report EPFL-REPORT-185242, 2013. 1.1.2

[86] T. Sheard. Using MetaML: A staged programming language. *Lecture Notes in Computer Science*, 1608, 1999. ISSN 0302-9743. 3

[87] T. Sheard and S.P. Jones. Template meta-programming for haskell. *ACM SIGPLAN Notices*, 37(12):60–75, 2002. 2.4.10, 2.4.10

[88] Tim Sheard and Nathan Linger. Programming in omega. In *Central European Functional Programming School, Second Summer School, CEFP 2007, Cluj-Napoca, 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

[89] Erik Silkensen and Jeremy G. Siek. Well-typed islands parse faster. In *Trends in Functional Programming - 13th International Symposium, TFP 2012, St. Andrews, UK, June 12-14, 2012, Revised Selected Papers*, pages 69–84, 2012. doi: 10.1007/978-3-642-40447-4_5. URL http://dx.doi.org/10.1007/978-3-642-40447-4_5. 2.4.8

[90] Konrad Slind. Object language embedding in Standard ML of New-Jersey. In *ICFP*, 1991. 2.4.2

[91] Eric Spishak, Werner Dietl, and Michael D Ernst. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs (FTfJP '12)*, pages 20–26, 2012. 4

[92] 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

[93] Christopher A Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic (TOCL)*, 7(4):676–722, 2006. 5.2.1

[94] Walid Taha and Patricia Johann. Staged notational definitions. In *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings*, pages 97–116, 2003. doi: 10.1007/978-3-540-39815-8_6. URL http://dx.doi.org/10.1007/978-3-540-39815-8_6. 2.4.4

[95] D. R. Tarditi and A. W. Appel. *ML-Yacc„version 2.0 Documentation*, 1990. 2.4.5

[96] Tim Teitelbaum and Thomas Reps. The Cornell Program Synthesizer: A Syntax-directed Programming Environment. *Commun. ACM*, 24(9):563–573, 1981. 2.4.8

[97] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968. ISSN 0001-0782. doi: 10.1145/363347.363387. URL http://doi.acm.org/10.1145/363347.363387. 2.3.2

[98] Arie van Deursen, Paul Klint, and Frank Tip. Origin tracking. *J. Symb. Comput.*, 15(5/6):523–545, 1993. doi: 10.1016/S0747-7171(06)80004-0. URL http://dx.doi.org/10.1016/S0747-7171(06)80004-0. 3.1.2

[99] E. Visser. Stratego: A language for program transformation based on rewriting strategies (system description). In A. Middeldorp, editor, *Rewriting Techniques and Applications, 12th International Conference, RTA-01*, LNCS 2051, pages 357–362, Utrecht, The Netherlands, May 22–24, 2001. Springer. 2.4.5

[100] 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

[101] Martin P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15 (4):147–161, 1994. 1.2.1

[102] Jacob Wieland. *Parsing Mixfix Expressions*. PhD thesis, Ph. D. thesis, Technische Universitat Berlin, 2009. 2.4.4

[103] Eric Van Wyk and August Schwerdfeger. Context-aware scanning for parsing extensible languages. In Charles Consel and Julia L. Lawall, editors, *GPCE*, pages 63–72. ACM, 2007. ISBN 978-1-59593-855-8. URL http://doi.acm.org/10.1145/1289971.1289983. 1.2.1, 1.2.1, 2.4.5, 2.4.8

Appendix A

Conventions

A.1 Typographic Conventions

We adopt *PFPL*'s typographic conventions for operational forms throughout the paper. For example, consider the operational form for injections into a labeled sum type:

...

In particular, the names of operators and indexed families of operators are written in `typewriter font`, indexed families of operators specify non-symbolic indices within [mathematical braces] and symbolic indices within [textual braces], and term arguments are grouped arbitrarily (roughly, by sort) using {textual curly braces} and (textual rounded braces) [47]. TODO: do we actually use symbols anymore?

Moreover, 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. $\mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})$, are identified up to mutual reordering of the label set and the corresponding argument sequence.

We write $\{r_i\}_{1 \leq i \leq n}$ for sequences of $n \geq 0$ rule arguments and $p.e$ for expressions binding the variables that appear in the pattern p.

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.

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.

A.2 Judgemental Conventions

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 syntactic forms, judgements, rules, proof clauses and proof cases typeset with 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
			$\texttt{fold}\{t.\tau\}(e)$	fold
			$\texttt{unfold}(e)$	unfold
			$\texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L})$	labeled tuple
			$\texttt{prj}[\ell](e)$	projection
			$\texttt{inj}[L;\ell]\{\{i \hookrightarrow \tau_i\}_{i\in L}\}(e)$	injection
			$\texttt{case}[L]\{\tau\}(e;\{i \hookrightarrow x_i.e_i\}_{i\in L})$	case analysis
			$\texttt{match}[n]\{\tau\}(e;\{r_i\}_{1\le i\le n})$	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

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 $x \in \mathrm{dom}(\Gamma)$, where $\mathrm{dom}(\Gamma)$ is a finite set of variables, to the hypothesis $x : \tau$, for some τ. We write $\Gamma, x : \tau$, when $x \notin \mathrm{dom}(\Gamma)$, for the extension of Γ with a mapping from x to $x : \tau$, and $\Gamma \cup \Gamma'$ when $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \emptyset$ for the typing context mapping each $x \in \mathrm{dom}(\Gamma) \cup \mathrm{dom}(\Gamma')$ 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

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau} \tag{B.2a}$$

$$\frac{\Delta \vdash \tau\ \mathsf{type} \qquad \Delta\ \Gamma, x : \tau \vdash e : \tau'}{\Delta\ \Gamma \vdash \mathtt{lam}\{\tau\}(x.e) : \mathtt{parr}(\tau; \tau')} \tag{B.2b}$$

$$\frac{\Delta\ \Gamma \vdash e_1 : \mathtt{parr}(\tau; \tau') \qquad \Delta\ \Gamma \vdash e_2 : \tau}{\Delta\ \Gamma \vdash \mathtt{ap}(e_1; e_2) : \tau'} \tag{B.2c}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \mathtt{tlam}(t.e) : \mathtt{all}(t.\tau)} \tag{B.2d}$$

$$\frac{\Delta\ \Gamma \vdash e : \mathtt{all}(t.\tau) \qquad \Delta \vdash \tau'\ \mathsf{type}}{\Delta\ \Gamma \vdash \mathtt{tap}\{\tau'\}(e) : [\tau'/t]\tau} \tag{B.2e}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type} \qquad \Delta\ \Gamma \vdash e : [\mathtt{rec}(t.\tau)/t]\tau}{\Delta\ \Gamma \vdash \mathtt{fold}\{t.\tau\}(e) : \mathtt{rec}(t.\tau)} \tag{B.2f}$$

$$\frac{\Delta\ \Gamma \vdash e : \mathtt{rec}(t.\tau)}{\Delta\ \Gamma \vdash \mathtt{unfold}(e) : [\mathtt{rec}(t.\tau)/t]\tau} \tag{B.2g}$$

$$\frac{\{\Delta\ \Gamma \vdash e_i : \tau_i\}_{i \in L}}{\Delta\ \Gamma \vdash \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) : \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{B.2h}$$

$$\frac{\Delta\ \Gamma \vdash e : \mathtt{prod}[L,\ell](\{i \hookrightarrow \tau_i\}_{i\in L}; \ell \hookrightarrow \tau)}{\Delta\ \Gamma \vdash \mathtt{prj}[\ell](e) : \tau} \tag{B.2i}$$

$$\frac{\{\Delta \vdash \tau_i\ \mathsf{type}\}_{i\in L} \qquad \Delta \vdash \tau\ \mathsf{type} \qquad \Delta\ \Gamma \vdash e : \tau}{\Delta\ \Gamma \vdash \mathtt{inj}[L,\ell;\ell]\{\{i \hookrightarrow \tau_i\}_{i\in L}; \ell \hookrightarrow \tau\}(e) : \mathtt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i\in L}; \ell \hookrightarrow \tau)} \tag{B.2j}$$

$$\frac{\Delta\ \Gamma \vdash e : \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i\in L}) \qquad \Delta \vdash \tau\ \mathsf{type} \qquad \{\Delta\ \Gamma, x_i : \tau_i \vdash e_i : \tau\}_{i\in L}}{\Delta\ \Gamma \vdash \mathtt{case}[L]\{\tau\}(e; \{i \hookrightarrow x_i.e_i\}_{i\in L}) : \tau} \tag{B.2k}$$

$$\frac{\Delta\ \Gamma \vdash e : \tau \qquad \Delta \vdash \tau'\ \mathsf{type} \qquad \{\Delta\ \Gamma \vdash r_i : \tau \Rightarrow\!\!\!\!\mapsto \tau'\}_{1\le i\le n}}{\Delta\ \Gamma \vdash \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1\le i\le n}) : \tau'} \tag{B.2l}$$

$\boxed{\Delta\ \Gamma \vdash r : \tau \Rightarrow\!\!\!\!\mapsto \tau'}$ r takes values of type τ to values of type τ'

$$\frac{\Delta \vdash p : \tau \dashv\!\mid \Gamma' \qquad \Delta\ \Gamma \cup \Gamma' \vdash e : \tau'}{\Delta\ \Gamma \vdash \mathtt{rule}(p.e) : \tau \Rightarrow\!\!\!\!\mapsto \tau'} \tag{B.3}$$

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

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

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

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

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

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

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). *All of the following hold:*

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* Δ, t type $\vdash p : \tau \dashv \Gamma$.

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). *All of the following hold:*

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). *All of the following hold:*

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 $\Delta\ [\tau'/t]\Gamma \vdash [\tau'/t]r : [\tau'/t]\tau \mapsto [\tau'/t]\tau''$ 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 $\Delta\ \Gamma \vdash [e'/x]r : \tau \mapsto \tau''$ does not depend on the form of e'.

□

The Pattern Regularity Lemma establishes that the hypotheses generated by checking a pattern against a well-formed type involve only well-formed types.

Lemma B.5 (Pattern Regularity). *If* $\Delta \vdash p : \tau \dashv\mid \Gamma$ *and* $\Delta \vdash \tau$ type *then* $\Delta \vdash \Gamma$ ctx.

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)

Case (B.4b).

(1)	$\Gamma = \emptyset$	by assumption
(2)	$\Delta \vdash \emptyset$ ctx	by Definition B.1

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)	$\{\Delta \vdash \tau_i$ type$\}_{i \in L}$	by Inversion of Rule (B.1e) on (5)
(7)	$\{\Delta \vdash \Gamma_i$ ctx$\}_{i \in L}$	by IH over (4) and (6)
(8)	$\Delta \vdash \cup_{i \in L} \Gamma_i$ ctx	by Definition B.1 on (7), then Definition B.1 again, using the definition of typing context union iteratively

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)

□

Finally, the Regularity Lemma establishes that the type assigned to an expression under a well-formed typing context is well-formed.

Lemma B.6 (Regularity). *All of the following hold:*

1. *If* $\Delta\ \Gamma \vdash e : \tau$ *and* $\Delta \vdash \Gamma$ ctx *then* $\Delta \vdash \tau$ type.
2. *If* $\Delta\ \Gamma \vdash r : \tau \Mapsto \tau'$ *and* $\Delta \vdash \Gamma$ ctx *then* $\Delta \vdash \tau'$ type.

Proof Sketch. By mutual rule induction over Rules (B.2) and Rule (B.3), and Lemma B.3 and Lemma B.5. □

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

Definition B.8 (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.9 (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* $e = \mathtt{fold}\{t.\tau'\}(e')$ *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* $e = \mathtt{inj}[L',\ell;\ell]\{\{i \hookrightarrow \tau_i\}_{i \in L'}; \ell \hookrightarrow \tau'\}(e')$ *and* $\vdash e' : \tau'$ *and* e' val.

Condition B.10 (Preservation). *If* $\vdash e : \tau$ *and* $e \mapsto e'$ *then* $\vdash e' : \tau$.

Condition B.11 (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 Abstract 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
			$\lambda \hat{x}{:}\hat{\tau}.\hat{e}$	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
			$\texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{by}\ \texttt{static}\ e\ \texttt{end}\ \texttt{in}\ \hat{e}$	seTSM definition
			$\hat{a}\ /b/$	seTSM application
			$\texttt{match}\ \hat{e}\ \{\hat{r}_i\}_{1 \leq i \leq n}$	match
			$\texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for patterns by}\ \texttt{static}\ e\ \texttt{end}\ \texttt{in}\ \hat{e}$	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 lengths of expression literal bodies in $\hat{e}$:

$$
\begin{array}{ll}
\|\hat{x}\| & = 0 \\
\|\lambda\hat{x}{:}\hat{\tau}.\hat{e}\| & = \|\hat{e}\| \\
\|\hat{e}_1(\hat{e}_2)\| & = \|\hat{e}_1\| + \|\hat{e}_2\| \\
\|\Lambda\hat{t}.\hat{e}\| & = \|\hat{e}\| \\
\|\hat{e}\texttt{[}\hat{\tau}\texttt{]}\| & = \|\hat{e}\| \\
\|\texttt{fold}(\hat{e})\| & = \|\hat{e}\| \\
\|\texttt{unfold}(\hat{e})\| & = \|\hat{e}\| \\
\|\langle\{i \hookrightarrow \hat{e}_i\}_{i\in L}\rangle\| & = \textstyle\sum_{i\in L} \|\hat{e}_i\| \\
\|\ell \cdot \hat{e}\| & = \|\hat{e}\| \\
\|\texttt{inj}[\ell](\hat{e})\| & = \|\hat{e}\| \\
\|\texttt{case}\ \hat{e}\ \{i \hookrightarrow \hat{x}_i.\hat{e}_i\}_{i\in L}\| & = \|\hat{e}\| + \textstyle\sum_{i\in L} \|\hat{e}_i\| \\
\|\texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{by static}\ e_{\text{parse}}\ \texttt{end in}\ \hat{e}\| & = \|\hat{e}\| \\
\|\hat{a}\ /b/\| & = \|b\| \\
\|\texttt{match}\ \hat{e}\ \{\hat{r}_i\}_{1\le i\le n}\| & = \|\hat{e}\| + \textstyle\sum_{1\le i\le n} \|r_i\| \\
\|\texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for patterns by static}\ e_{\text{parse}}\ \texttt{end in}\ \hat{e}\| & = \|\hat{e}\|
\end{array}
$$

and $\|r\|$ is defined as follows:

$$\|\hat{p} \Rightarrow \hat{e}\| = \|\hat{e}\|$$

Similarly, the metafunction $\|\hat{p}\|$ computes the sum of the lengths of the pattern literal bodies in $\hat{p}$:

$$
\begin{aligned}
\|\hat{x}\| &= 0 \\
\|\texttt{fold(}\hat{p}\texttt{)}\| &= \|\hat{p}\| \\
\|\langle L\rangle\{i \hookrightarrow \hat{p}_i\}_{i\in L}\| &= \sum_{i\in L} \|\hat{p}_i\| \\
\|\texttt{inj}[\ell](\hat{p})\| &= \|\hat{p}\| \\
\|\hat{a}\ /b/\| &= \|b\|
\end{aligned}
$$

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:

$$
\begin{aligned}
\mathcal{U}(t) &= \widehat{t} \\
\mathcal{U}(\texttt{parr(}\tau_1\texttt{;}\tau_2\texttt{)}) &= \mathcal{U}(\tau_1) \rightharpoonup \mathcal{U}(\tau_2) \\
\mathcal{U}(\texttt{all(}t\texttt{.}\tau\texttt{)}) &= \forall\widehat{t}.\mathcal{U}(\tau) \\
\mathcal{U}(\texttt{rec(}t\texttt{.}\tau\texttt{)}) &= \mu\widehat{t}.\mathcal{U}(\tau)
\end{aligned}
$$

$$\mathcal{U}(\texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})) = \langle \{i \hookrightarrow \mathcal{U}(\tau_i)\}_{i \in L} \rangle$$
$$\mathcal{U}(\texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L})) = [\{i \hookrightarrow \mathcal{U}(\tau_i)\}_{i \in L}]$$

- Each expression variable, x, maps onto a unique expression identifier, written $\widehat{x}$.
- Each expanded expression form, e, maps onto an unexpanded expression form $\mathcal{U}(e)$ as follows:

$$\begin{aligned}
\mathcal{U}(x) &= \widehat{x} \\
\mathcal{U}(\texttt{lam}\{\tau\}(x.e)) &= \lambda \widehat{x}{:}\mathcal{U}(\tau).\mathcal{U}(e) \\
\mathcal{U}(\texttt{ap}(e_1; e_2)) &= \mathcal{U}(e_1)(\mathcal{U}(e_2)) \\
\mathcal{U}(\texttt{tlam}(t.e)) &= \Lambda \widehat{t}.\mathcal{U}(e) \\
\mathcal{U}(\texttt{tap}\{\tau\}(e)) &= \mathcal{U}(e)[\mathcal{U}(\tau)] \\
\mathcal{U}(\texttt{fold}\{t.\tau\}(e)) &= \texttt{fold}(\mathcal{U}(e)) \\
\mathcal{U}(\texttt{unfold}(e)) &= \texttt{unfold}(\mathcal{U}(e)) \\
\mathcal{U}(\texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L})) &= \langle \{i \hookrightarrow \mathcal{U}(e_i)\}_{i \in L} \rangle \\
\mathcal{U}(\texttt{inj}[L; \ell]\{\{i \hookrightarrow \tau_i\}_{i \in L}\}(e)) &= \texttt{inj}[\ell](\mathcal{U}(e)) \\
\mathcal{U}(\texttt{match}[n]\{\tau\}(e; \{r_i\}_{1 \le i \le n})) &= \texttt{match}\,\mathcal{U}(e)\ \{\mathcal{U}(r_i)\}_{1 \le i \le n}
\end{aligned}$$

- The expanded rule form maps onto the unexpanded rule form as follows:

$$\mathcal{U}(\texttt{rule}(p.e)) = \texttt{urule}(\mathcal{U}(p).\mathcal{U}(e))$$

- Each expanded pattern form, p, maps onto the unexpanded pattern form $\mathcal{U}(p)$ 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 given in Figure 3.6, there is also a context-free textual syntax for the UL. We need only posit the existence of partial metafunctions $\mathsf{parseUTyp}(b)$ and $\mathsf{parseUExp}(b)$ and $\mathsf{parseUPat}(b)$.

Condition B.12 (Textual Representability). *All of the following must hold:*

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

Condition B.14 (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 $\hat{t} \in \mathrm{dom}(\mathcal{D})$ 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 $\hat{\Delta}, \hat{t} \rightsquigarrow t\ \mathsf{type}$ 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.15 (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

Unexpanded Typing 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 $\hat{x} \in \mathrm{dom}(\mathcal{G})$ 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

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

Definition B.16 (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* $x \in dom(\Gamma)$.

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 $\hat{a} \in \mathrm{dom}(\mathcal{A})$ 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 $a \in \mathrm{dom}(\Psi)$ 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 $a \notin \mathrm{dom}(\Psi)$ for the extension of Ψ that maps a to $a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}})$. We write $\Delta \vdash \Psi$ seTSMs when all the type annotations in Ψ are well-formed assuming Δ, and the parse functions in Ψ are closed and of type $\texttt{Body} \rightharpoonup \texttt{ParseResultSE}$.

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

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

Definition B.17 (seTSM Definition Context Formation). $\Delta \vdash \Psi$ seTSMs *iff for each* $\hat{a} \hookrightarrow \texttt{setsm}(\tau; e_{parse}) \in \Psi$, *we have* $\Delta \vdash \tau$ type *and* $\emptyset\,\emptyset \vdash e_{parse} : \texttt{Body} \rightharpoonup \texttt{ParseResultSE}$.
Definition B.18 (seTSM Context Formation). $\Delta \vdash \langle \mathcal{A}; \Psi \rangle$ seTSMctx *iff* $\Delta \vdash \Psi$ seTSMs *and for each* $\hat{a} \rightsquigarrow a \in \mathcal{A}$ *we have* $a \in dom(\Psi)$.

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 Ψ is a *spTSM definition context*.

An *spTSM definition context*, Φ, is a finite function mapping each TSM name $a \in \mathrm{dom}(\Phi)$ 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 $a \notin \mathrm{dom}(\Phi)$ for the extension of Φ that maps a to $a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}})$. We write $\Delta \vdash \Phi$ spTSMs when all the type annotations in Φ are well-formed assuming Δ, and the parse functions in Φ are closed and of type $\texttt{Body} \rightharpoonup \texttt{ParseResultSP}$.

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

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

Definition B.19 (spTSM Definition Context Formation). $\Delta \vdash \Phi$ spTSMs *iff for each* $\hat{a} \hookrightarrow \texttt{sptsm}(\tau; e_{parse}) \in \Phi$, *we have* $\Delta \vdash \tau$ type *and* $\emptyset\,\emptyset \vdash e_{parse} : \texttt{Body} \rightharpoonup \texttt{ParseResultSP}$.
Definition B.20 (spTSM Context Formation). $\Delta \vdash \langle \mathcal{A}; \Phi \rangle$ spTSMctx *iff* $\Delta \vdash \Phi$ spTSMs *and for each* $\hat{a} \rightsquigarrow a \in \mathcal{A}$ *we have* $a \in dom(\Phi)$.

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* $b \downarrow_{\mathsf{Body}} e_{\mathrm{body}}$. An inverse mapping is defined by the *body decoding judgement* $e_{\mathrm{body}} \uparrow_{\mathsf{Body}} b$.

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.21 (Body Isomorphism)**.** *All of the following must hold:*

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 B.22 (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:

$$L_{\mathtt{SE}} \stackrel{\text{def}}{=} \mathtt{ParseError}, \mathtt{SuccessE}$$
$$\mathtt{ParseResultSE} \stackrel{\text{def}}{=} \mathtt{sum}[L_{\mathtt{SE}}](\mathtt{ParseError} \hookrightarrow \langle\rangle, \mathtt{SuccessE} \hookrightarrow \mathtt{ProtoExpr})$$
$$\mathtt{SuccessE} \cdot e \stackrel{\text{def}}{=} \mathtt{inj}[L_{\mathtt{SE}}; \mathtt{SuccessE}]\{\mathtt{ParseError} \hookrightarrow \langle\rangle, \mathtt{SuccessE} \hookrightarrow \mathtt{ProtoExpr}\}(e)$$

The type abbreviated `ParseResultSP`, and an auxiliary abbreviation used below, is defined as follows:

$$L_{\mathtt{SP}} \stackrel{\text{def}}{=} \mathtt{ParseError}, \mathtt{SuccessP}$$
$$\mathtt{ParseResultSE} \stackrel{\text{def}}{=} \mathtt{sum}[L_{\mathtt{SP}}](\mathtt{ParseError} \hookrightarrow \langle\rangle, \mathtt{SuccessP} \hookrightarrow \mathtt{ProtoPat})$$
$$\mathtt{SuccessP} \cdot e \stackrel{\text{def}}{=} \mathtt{inj}[L_{\mathtt{SP}}; \mathtt{SuccessP}]\{\mathtt{ParseError} \hookrightarrow \langle\rangle, \mathtt{SuccessP} \hookrightarrow \mathtt{ProtoPat}\}(e)$$

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

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

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

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

$$\frac{\hat{\Delta}, \hat{t} \rightsquigarrow t\ \mathsf{type}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \Lambda \hat{t}.\hat{e} \rightsquigarrow \mathtt{tlam}(t.e) : \mathtt{all}(t.\tau)} \tag{B.6d}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \mathtt{all}(t.\tau) \qquad \hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'\ \mathsf{type}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}[\hat{\tau}'] \rightsquigarrow \mathtt{tap\{\tau'\}(e)} : [\tau'/t]\tau} \tag{B.6e}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type} \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : [\mathtt{rec}(t.\tau)/t]\tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{fold}(\hat{e}) \rightsquigarrow \mathtt{fold\{t.\tau\}(e)} : \mathtt{rec}(t.\tau)} \tag{B.6f}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \mathtt{rec}(t.\tau)}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{unfold}(\hat{e}) \rightsquigarrow \mathtt{unfold}(e) : [\mathtt{rec}(t.\tau)/t]\tau} \tag{B.6g}$$

$$\frac{\{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_i \rightsquigarrow e_i : \tau_i\}_{i \in L}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \langle \{i \hookrightarrow \hat{e}_i\}_{i \in L} \rangle \rightsquigarrow \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) : \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{B.6h}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \mathtt{prod}[L,\ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \cdot \ell \rightsquigarrow \mathtt{prj}[\ell](e) : \tau} \tag{B.6i}$$

$$\frac{\{\Delta \vdash \tau_i\ \mathsf{type}\}_{i \in L} \qquad \Delta \vdash \tau\ \mathsf{type} \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau}{\left(\begin{array}{c} \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Phi};\hat{\Psi}} \mathtt{inj}[\ell](\hat{e}) \\ \rightsquigarrow \\ \mathtt{inj}[L,\ell;\ell]\{\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau\}(e) : \mathtt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \end{array}\right)} \tag{B.6j}$$

$$\frac{\begin{array}{c} \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \\ \Delta \vdash \tau\ \mathsf{type} \qquad \{\hat{\Delta}\ \hat{\Gamma}, \hat{x}_i \rightsquigarrow x_i : \tau_i \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_i \rightsquigarrow e_i : \tau\}_{i \in L} \end{array}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{case}\ \hat{e}\ \{i \hookrightarrow \hat{x}_i.\hat{e}_i\}_{i \in L} \rightsquigarrow \mathtt{case}[L]\{\tau\}(e; \{i \hookrightarrow x_i.e_i\}_{i \in L}) : \tau} \tag{B.6k}$$

$$\frac{\begin{array}{c} \hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \varnothing\ \varnothing \vdash e_{\mathrm{parse}} : \mathtt{parr}(\mathtt{Body};\mathtt{ParseResultSE}) \\ \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi},\hat{a} \rightsquigarrow a \hookrightarrow \mathtt{setsm}(\tau;e_{\mathrm{parse}});\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau' \end{array}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{by}\ \mathtt{static}\ e_{\mathrm{parse}}\ \mathtt{end}\ \mathtt{in}\ \hat{e} \rightsquigarrow e : \tau'} \tag{B.6l}$$

$$\frac{\begin{array}{c}\hat{\Psi} = \hat{\Psi}', \hat{a} \rightsquigarrow a \hookrightarrow \texttt{setsm}(\tau; e_{\text{parse}}) \\ b \downarrow_{\text{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{SuccessE} \cdot e_{\text{proto}} \qquad e_{\text{proto}} \uparrow_{\text{ProtoExpr}} \grave{e} \\ \text{seg}(\grave{e}) \text{ segments } b \qquad \varnothing\,\varnothing \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e : \tau\end{array}}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{a}\ /b/ \rightsquigarrow e : \tau} \tag{B.6m}$$

$$\frac{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Phi};\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau \qquad \Delta \vdash \tau' \text{ type} \qquad \{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \rightsquigarrow r_i : \tau \Mapsto \tau'\}_{1 \le i \le n}}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{match}\ \hat{e}\ \{\hat{r}_i\}_{1 \le i \le n} \rightsquigarrow \texttt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \le i \le n}) : \tau'} \tag{B.6n}$$

$$\frac{\begin{array}{c}\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \text{ type} \qquad \varnothing\,\varnothing \vdash e_{\text{parse}} : \texttt{parr}(\texttt{Body}; \texttt{ParseResultSP}) \\ \hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi},\hat{a} \rightsquigarrow a \hookrightarrow \texttt{sptsm}(\tau; e_{\text{parse}})} \hat{e} \rightsquigarrow e : \tau'\end{array}}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for patterns by static}\ e_{\text{parse}}\ \texttt{end in}\ \hat{e} \rightsquigarrow e : \tau'} \tag{B.6o}$$

$\boxed{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \rightsquigarrow r : \tau \Mapsto \tau'}$ $\hat{r}$ has expansion r taking values of type τ to values of type τ'

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

Rule (B.7) is defined mutually with Rules (B.6).

Typed Pattern Expansion

$\boxed{\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\vdash \hat{\Gamma}}$ $\hat{p}$ has expansion p matching against τ generating hypotheses $\hat{\Gamma}$

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

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

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

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

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

$$\frac{\begin{array}{c}\hat{\Phi} = \hat{\Phi}', \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{sptsm}(\tau; e_{\text{parse}}) \\ b \downarrow_{\text{Body}} e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \mathtt{SuccessP} \cdot e_{\text{proto}} \qquad e_{\text{proto}} \uparrow_{\text{ProtoPat}} \grave{p} \\ \text{seg}(\grave{p}) \text{ segments } b \qquad \vdash^{\Delta;\hat{\Phi};b} \grave{p} \rightsquigarrow p : \tau \dashv\!|^{\hat{\Gamma}}\end{array}}{\Delta \vdash_{\hat{\Phi}} \hat{a}\ /b/ \rightsquigarrow p : \tau \dashv\!| \hat{\Gamma}} \tag{B.8f}$$

In Rule (B.8d), $\hat{\Gamma}_i$ is shorthand for $\langle \mathcal{G}_i; \Gamma_i \rangle$ and $\cup_{i \in L} \hat{\Gamma}_i$ is shorthand for

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

B.3 Proto-Expansion Validation

B.3.1 Syntax of Proto-Expansions

Sort		Operational Form	Stylized Form	Description
PrTyp	$\grave{\tau}$ ::=	t	t	variable
		$\mathtt{prparr}(\grave{\tau}; \grave{\tau})$	$\grave{\tau} \rightharpoonup \grave{\tau}$	partial function
		$\mathtt{prall}(t.\grave{\tau})$	$\forall t.\grave{\tau}$	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
		$\mathtt{splicedt}[m; n]$	$\mathtt{splicedt}\langle m, n \rangle$	spliced
PrExp	$\grave{e}$ ::=	x	x	variable
		$\mathtt{prlam}\{\grave{\tau}\}(x.\grave{e})$	$\lambda x{:}\grave{\tau}.\grave{e}$	abstraction
		$\mathtt{prap}(\grave{e}; \grave{e})$	$\grave{e}(\grave{e})$	application
		$\mathtt{prtlam}(t.\grave{e})$	$\Lambda t.\grave{e}$	type abstraction
		$\mathtt{prtap}\{\grave{\tau}\}(\grave{e})$	$\grave{e}[\grave{\tau}]$	type application
		$\mathtt{prfold}\{t.\grave{\tau}\}(\grave{e})$	$\mathtt{fold}(\grave{e})$	fold
		$\mathtt{prunfold}(\grave{e})$	$\mathtt{unfold}(\grave{e})$	unfold
		$\mathtt{prtpl}\{L\}(\{i \hookrightarrow \grave{e}_i\}_{i \in L})$	$\langle \{i \hookrightarrow \grave{e}_i\}_{i \in L} \rangle$	labeled tuple
		$\mathtt{prprj}[\ell](\grave{e})$	$\grave{e} \cdot \ell$	projection
		$\mathtt{prinj}[L; \ell]\{\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}\}(\grave{e})$	$\mathtt{inj}[\ell](\grave{e})$	injection
		$\mathtt{prcase}[L]\{\grave{\tau}\}(\grave{e}; \{i \hookrightarrow x_i.\grave{e}_i\}_{i \in L})$	$\mathtt{case}\ \grave{e}\ \{i \hookrightarrow x_i.\grave{e}_i\}_{i \in L}$	case analysis
		$\mathtt{splicede}[m; n]$	$\mathtt{splicede}\langle m, n \rangle$	spliced
		$\mathtt{prmatch}[n]\{\grave{\tau}\}(\grave{e}; \{\grave{r}_i\}_{1 \le i \le n})$	$\mathtt{match}\ \grave{e}\ \{\grave{r}_i\}_{1 \le i \le n}$	match
PrRule	$\grave{r}$ ::=	$\mathtt{prrule}(p.\grave{e})$	$p \Rightarrow \grave{e}$	rule
PrPat	$\grave{p}$ ::=	$\mathtt{prwildp}$	$\_$	wildcard pattern
		$\mathtt{prfoldp}(p)$	$\mathtt{fold}(p)$	fold pattern
		$\mathtt{prtplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i \in L})$	$\langle \{i \hookrightarrow \grave{p}_i\}_{i \in L} \rangle$	labeled tuple pattern
		$\mathtt{prinjp}[\ell](\grave{p})$	$\mathtt{inj}[\ell](\grave{p})$	injection pattern
		$\mathtt{splicedp}[m; n]$	$\mathtt{splicedp}\langle m, n \rangle$	spliced

Common Proto-Expansion Forms

Each expanded form, except the variable pattern form, maps onto a proto-expansion form. We refer to these as the *common proto-expansion forms*. In particular:

- Each type form, τ, maps onto a proto-type form, $\mathcal{P}(\tau)$, as follows:

$$\begin{aligned}
\mathcal{P}(t) &= t \\
\mathcal{P}(\texttt{parr(}\tau_1\texttt{;}\tau_2\texttt{)}) &= \texttt{prparr(}\mathcal{P}(\tau_1)\texttt{;}\mathcal{P}(\tau_2)\texttt{)} \\
\mathcal{P}(\texttt{all(}t\texttt{.}\tau\texttt{)}) &= \texttt{prall(}t\texttt{.}\mathcal{P}(\tau)\texttt{)} \\
\mathcal{P}(\texttt{rec(}t\texttt{.}\tau\texttt{)}) &= \texttt{prrec(}t\texttt{.}\mathcal{P}(\tau)\texttt{)} \\
\mathcal{P}(\texttt{prod}[L]\texttt{(}\{i \hookrightarrow \tau_i\}_{i\in L}\texttt{)}) &= \texttt{prprod}[L]\texttt{(}\{i \hookrightarrow \mathcal{P}(\tau_i)\}_{i\in L}\texttt{)} \\
\mathcal{P}(\texttt{sum}[L]\texttt{(}\{i \hookrightarrow \tau_i\}_{i\in L}\texttt{)}) &= \texttt{prsum}[L]\texttt{(}\{i \hookrightarrow \mathcal{P}(\tau_i)\}_{i\in L}\texttt{)}
\end{aligned}$$

- Each expanded expression form, e, maps onto a proto-expression, $\mathcal{P}(e)$, as follows:

$$\begin{aligned}
\mathcal{P}(x) &= x \\
\mathcal{P}(\texttt{lam\{}\tau\texttt{\}(}x\texttt{.}e\texttt{)}) &= \texttt{prlam\{}\mathcal{P}(\tau)\texttt{\}(}x\texttt{.}\mathcal{P}(e)\texttt{)} \\
\mathcal{P}(\texttt{ap(}e_1\texttt{;}e_2\texttt{)}) &= \texttt{prap(}\mathcal{P}(e_1)\texttt{;}\mathcal{P}(e_2)\texttt{)} \\
\mathcal{P}(\texttt{tlam(}t\texttt{.}e\texttt{)}) &= \texttt{prtlam(}t\texttt{.}\mathcal{P}(e)\texttt{)} \\
\mathcal{P}(\texttt{tap\{}\tau\texttt{\}(}e\texttt{)}) &= \texttt{prtap\{}\mathcal{P}(\tau)\texttt{\}(}\mathcal{P}(e)\texttt{)} \\
\mathcal{P}(\texttt{fold\{}t\texttt{.}\tau\texttt{\}(}e\texttt{)}) &= \texttt{prfold\{}t\texttt{.}\mathcal{P}(\tau)\texttt{\}(}\mathcal{P}(e)\texttt{)} \\
\mathcal{P}(\texttt{unfold(}e\texttt{)}) &= \texttt{prunfold(}\mathcal{P}(e)\texttt{)} \\
\mathcal{P}(\texttt{tpl}[L]\texttt{(}\{i \hookrightarrow e_i\}_{i\in L}\texttt{)}) &= \texttt{prtpl\{}L\texttt{\}(}\{i \hookrightarrow \mathcal{P}(e_i)\}_{i\in L}\texttt{)} \\
\mathcal{P}(\texttt{inj}[L\texttt{;}\ell]\texttt{\{}\{i \hookrightarrow \tau_i\}_{i\in L}\texttt{\}(}e\texttt{)}) &= \texttt{prinj}[L\texttt{;}\ell]\texttt{\{}\{i \hookrightarrow \mathcal{P}(\tau_i)\}_{i\in L}\texttt{\}(}\mathcal{P}(e)\texttt{)} \\
\mathcal{P}(\texttt{match}[n]\texttt{\{}\tau\texttt{\}(}e\texttt{;}\{r_i\}_{1\le i\le n}\texttt{)}) &= \texttt{prmatch}[n]\texttt{\{}\mathcal{P}(\tau)\texttt{\}(}\mathcal{P}(e)\texttt{;}\{\mathcal{P}(r_i)\}_{1\le i\le n}\texttt{)}
\end{aligned}$$

- The expanded rule form maps onto the proto-rule form as follows:

$$\mathcal{P}(\texttt{rule(}p\texttt{.}e\texttt{)}) = \texttt{prrule(}p\texttt{.}\mathcal{P}(e)\texttt{)}$$

  Notice that proto-rules bind expanded patterns, not proto-patterns. This is because proto-rules appear in protob-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 form, p, except for the variable pattern form, maps onto a proto-pattern form, $\mathcal{P}(p)$, as follows:

$$\begin{aligned}
\mathcal{P}(\texttt{wildp}) &= \texttt{prwildp} \\
\mathcal{P}(\texttt{foldp(}p\texttt{)}) &= \texttt{prfoldp(}\mathcal{P}(p)\texttt{)} \\
\mathcal{P}(\texttt{tplp}[L]\texttt{(}\{i \hookrightarrow p_i\}_{i\in L}\texttt{)}) &= \texttt{prtplp}[L]\texttt{(}\{i \hookrightarrow \mathcal{P}(p_i)\}_{i\in L}\texttt{)} \\
\mathcal{P}(\texttt{injp}[\ell]\texttt{(}p\texttt{)}) &= \texttt{prinjp}[\ell]\texttt{(}\mathcal{P}(p)\texttt{)}
\end{aligned}$$

Proto-Expression Encoding and Decoding

The type abbreviated `ProtoExpr` classifies encodings of *proto-expressions*. The mapping from proto-expressions to values of type `ProtoExpr` is defined by the *proto-expression*

encoding judgement, $\grave{e} \downarrow_{\mathsf{ProtoExpr}} e$. An inverse mapping is defined by the *proto-expression decoding judgement*, $e \uparrow_{\mathsf{ProtoExpr}} \grave{e}$.

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

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

Condition B.23 (Proto-Expression Isomorphism)**.** *All of the following must hold:*

1. *For every* $\grave{e}$*, we have* $\grave{e} \downarrow_{\mathsf{ProtoExpr}} e_{proto}$ *for some* e_{proto} *such that* $\vdash e_{proto}$: `ProtoExpr` *and* e_{proto} val.
2. *If* $\vdash e_{proto}$: `ProtoExpr` *and* e_{proto} val *then* $e_{proto} \uparrow_{\mathsf{ProtoExpr}} \grave{e}$ *for some* $\grave{e}$*.*
3. *If* $\grave{e} \downarrow_{\mathsf{ProtoExpr}} e_{proto}$ *then* $e_{proto} \uparrow_{\mathsf{ProtoExpr}} \grave{e}$*.*
4. *If* $\vdash e_{proto}$: `ProtoExpr` *and* e_{proto} val *and* $e_{proto} \uparrow_{\mathsf{ProtoExpr}} \grave{e}$ *then* $\grave{e} \downarrow_{\mathsf{ProtoExpr}} e_{proto}$*.*
5. *If* $\grave{e} \downarrow_{\mathsf{ProtoExpr}} e_{proto}$ *and* $\grave{e} \downarrow_{\mathsf{ProtoExpr}} e'_{proto}$ *then* $e_{proto} = e'_{proto}$*.*
6. *If* $\vdash e_{proto}$: `ProtoExpr` *and* e_{proto} val *and* $e_{proto} \uparrow_{\mathsf{ProtoExpr}} \grave{e}$ *and* $e_{proto} \uparrow_{\mathsf{ProtoExpr}} \grave{e}'$ *then* $\grave{e} = \grave{e}'$*.*

Proto-Pattern Encoding and Decoding

The type abbreviated `ProtoPat` classifies encodings of *proto-patterns*. The mapping from proto-patterns to values of type `ProtoPat` is defined by the *proto-pattern encoding judgement*, $\grave{p} \downarrow_{\mathsf{ProtoPat}} p$. An inverse mapping is defined by the *proto-expression decoding judgement*, $p \uparrow_{\mathsf{ProtoPat}} \grave{p}$.

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

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

Condition B.24 (Proto-Pattern Isomorphism)**.** *All of the following must hold:*

1. *For every* $\grave{p}$*, we have* $\grave{p} \downarrow_{\mathsf{ProtoPat}} e_{proto}$ *for some* e_{proto} *such that* $\vdash e_{proto}$: `ProtoPat` *and* e_{proto} val.
2. *If* $\vdash e_{proto}$: `ProtoPat` *and* e_{proto} val *then* $e_{proto} \uparrow_{\mathsf{ProtoPat}} \grave{p}$ *for some* $\grave{p}$*.*
3. *If* $\grave{p} \downarrow_{\mathsf{ProtoPat}} e_{proto}$ *then* $e_{proto} \uparrow_{\mathsf{ProtoPat}} \grave{p}$*.*
4. *If* $\vdash e_{proto}$: `ProtoPat` *and* e_{proto} val *and* $e_{proto} \uparrow_{\mathsf{ProtoPat}} \grave{p}$ *then* $\grave{p} \downarrow_{\mathsf{ProtoPat}} e_{proto}$*.*
5. *If* $\grave{p} \downarrow_{\mathsf{ProtoPat}} e_{proto}$ *and* $\grave{p} \downarrow_{\mathsf{ProtoPat}} e'_{proto}$ *then* $e_{proto} = e'_{proto}$*.*
6. *If* $\vdash e_{proto}$: `ProtoPat` *and* e_{proto} val *and* $e_{proto} \uparrow_{\mathsf{ProtoPat}} \grave{p}$ *and* $e_{proto} \uparrow_{\mathsf{ProtoPat}} \grave{p}'$ *then* $\grave{p} = \grave{p}'$*.*

Segmentations

A *segmentation*, ψ, is a finite set of *segments*. Segments consist of two natural numbers and a sort, i.e. segments are of the form $\langle m; n; \mathsf{UExp}\rangle$ or $\langle m; n; \mathsf{UTyp}\rangle$ or $\langle m; n; \mathsf{UPat}\rangle$.

The metafunction $\mathsf{seg}(\grave{e})$ determines the segmentation of $\grave{e}$ by generating one segment for each reference to a spliced expression or type, respectively. More specifically:

- We define $\mathsf{seg}(\grave{\tau})$ as follows:

$$\begin{aligned}
\mathsf{seg}(t) &= \emptyset \\
\mathsf{seg}(\texttt{prparr(}\grave{\tau}_1\texttt{;}\grave{\tau}_2\texttt{)}) &= \mathsf{seg}(\grave{\tau}_1) \cup \mathsf{seg}(\grave{\tau}_2) \\
\mathsf{seg}(\texttt{prall(}t\texttt{.}\grave{\tau}\texttt{)}) &= \mathsf{seg}(\grave{\tau}) \\
\mathsf{seg}(\texttt{prrec(}t\texttt{.}\grave{\tau}\texttt{)}) &= \mathsf{seg}(\grave{\tau}) \\
\mathsf{seg}(\texttt{prprod}[L]\texttt{(}\{i \hookrightarrow \grave{\tau}_i\}_{i\in L}\texttt{)}) &= \cup_{i\in L}\mathsf{seg}(\grave{\tau}_i) \\
\mathsf{seg}(\texttt{prsum}[L]\texttt{(}\{i \hookrightarrow \grave{\tau}_i\}_{i\in L}\texttt{)}) &= \cup_{i\in L}\mathsf{seg}(\grave{\tau}_i) \\
\mathsf{seg}(\texttt{splicedt}[m; n]) &= \{\langle m; n; \mathsf{UTyp}\rangle\}
\end{aligned}$$

- We define $\mathsf{seg}(\grave{e})$ as follows:

$$\begin{aligned}
\mathsf{seg}(x) &= \emptyset \\
\mathsf{seg}(\texttt{prlam\{}\grave{\tau}\texttt{\}(}x\texttt{.}\grave{e}\texttt{)}) &= \mathsf{seg}(\grave{\tau}) \cup \mathsf{seg}(\grave{e}) \\
\mathsf{seg}(\texttt{prtlam(}t\texttt{.}\grave{e}\texttt{)}) &= \mathsf{seg}(\grave{e}) \\
\mathsf{seg}(\texttt{prtap\{}\grave{\tau}\texttt{\}(}\grave{e}\texttt{)}) &= \mathsf{seg}(\grave{\tau}) \cup \mathsf{seg}(\grave{e}) \\
\mathsf{seg}(\texttt{prfold\{}t\texttt{.}\grave{\tau}\texttt{\}(}\grave{e}\texttt{)}) &= \mathsf{seg}(\grave{\tau}) \cup \mathsf{seg}(\grave{e}) \\
\mathsf{seg}(\texttt{prunfold(}\grave{e}\texttt{)}) &= \mathsf{seg}(\grave{e}) \\
\mathsf{seg}(\texttt{prtpl\{}L\texttt{\}(}\{i \hookrightarrow \grave{e}_i\}_{i\in L}\texttt{)}) &= \cup_{i\in L}\mathsf{seg}(\grave{e}_i) \\
\mathsf{seg}(\texttt{prprj}[\ell]\texttt{(}\grave{e}\texttt{)}) &= \mathsf{seg}(\grave{e}) \\
\mathsf{seg}(\texttt{prinj}[L; \ell]\texttt{\{}\{i \hookrightarrow \grave{\tau}_i\}_{i\in L}\texttt{\}(}\grave{e}\texttt{)}) &= \cup_{i\in L}\mathsf{seg}(\tau_i) \cup \mathsf{seg}(\grave{e}) \\
\mathsf{seg}(\texttt{prcase}[L]\texttt{\{}\grave{\tau}\texttt{\}(}\grave{e}\texttt{;}\{i \hookrightarrow x_i\texttt{.}\grave{e}_i\}_{i\in L}\texttt{)}) &= \mathsf{seg}(\grave{\tau}) \cup \mathsf{seg}(\grave{e}) \cup_{i\in L} \mathsf{seg}(\grave{e}_i) \\
\mathsf{seg}(\texttt{splicede}[m; n]) &= \{\langle m; n; \mathsf{UExp}\rangle\} \\
\mathsf{seg}(\texttt{prmatch}[n]\texttt{\{}\grave{\tau}\texttt{\}(}\grave{e}\texttt{;}\{\grave{r}_i\}_{1\le i\le n}\texttt{)}) &= \mathsf{seg}(\grave{\tau}) \cup \mathsf{seg}(\grave{e}) \cup_{1\le i\le n} \mathsf{seg}(\grave{r}_i)
\end{aligned}$$

- We define $\mathsf{seg}(\grave{r})$ as follows:

$$\mathsf{seg}(\texttt{prrule(}p\texttt{.}\grave{e}\texttt{)}) = \mathsf{seg}(\grave{e})$$

The metafunction $\mathsf{seg}(\grave{p})$ determines the segmentation of $\grave{p}$ by generating one segment for each reference to a spliced pattern:

$$\begin{aligned}
\mathsf{seg}(\texttt{prwildp}) &= \emptyset \\
\mathsf{seg}(\texttt{prfoldp(}\grave{p}\texttt{)}) &= \mathsf{seg}(\grave{p}) \\
\mathsf{seg}(\texttt{prtplp}[L]\texttt{(}\{i \hookrightarrow \grave{p}_i\}_{i\in L}\texttt{)}) &= \cup_{i\in L}\mathsf{seg}(\grave{p}_i) \\
\mathsf{seg}(\texttt{prinjp}[\ell]\texttt{(}\grave{p}\texttt{)}) &= \mathsf{seg}(\grave{p}) \\
\mathsf{seg}(\texttt{splicedp}[m; n]) &= \{\langle m; n; \mathsf{UPat}\rangle\}
\end{aligned}$$

The predicate ψ 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 $\hat{\Delta}; b$.

$\boxed{\Delta \vdash^{\mathbb{T}} \grave{\tau} \rightsquigarrow \tau \text{ type}}$ $\grave{\tau}$ has well-formed expansion τ

$$\frac{}{\Delta, t \text{ type} \vdash^{\mathbb{T}} t \rightsquigarrow t \text{ type}} \tag{B.9a}$$

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

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

$$\frac{\Delta, t \text{ type} \vdash^{\mathbb{T}} \grave{\tau} \rightsquigarrow \tau \text{ type}}{\Delta \vdash^{\mathbb{T}} \mathtt{prrec}(t.\grave{\tau}) \rightsquigarrow \mathtt{rec}(t.\tau) \text{ type}} \tag{B.9d}$$

$$\frac{\{\Delta \vdash^{\mathbb{T}} \grave{\tau}_i \rightsquigarrow \tau_i \text{ type}\}_{i \in L}}{\Delta \vdash^{\mathbb{T}} \mathtt{prprod}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}) \rightsquigarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \text{ type}} \tag{B.9e}$$

$$\frac{\{\Delta \vdash^{\mathbb{T}} \grave{\tau}_i \rightsquigarrow \tau_i \text{ type}\}_{i \in L}}{\Delta \vdash^{\mathbb{T}} \mathtt{prsum}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}) \rightsquigarrow \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \text{ type}} \tag{B.9f}$$

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

The following lemma establishes that each type can be expressed as a well-formed proto-type, under the same type formation context and any type splicing scene.

Lemma B.25 (Proto-Expansion Type Expressibility)**.** *If* $\Delta \vdash \tau \text{ type}$ *then* $\Delta \vdash^{\mathbb{T}} \mathcal{P}(\tau) \rightsquigarrow \tau \text{ type}$.

Proof. By rule induction over Rules (B.1). In each case, we apply the IH on or over each premise, then apply the corresponding proto-type validation rule in Rules (B.9). □

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 $\mathbb{E}$:

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

$\boxed{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau}$ $\grave{e}$ has expansion e of type τ

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

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

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

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prtlam}(t.\grave{e}) \rightsquigarrow \mathtt{tlam}(t.e) : \mathtt{all}(t.\tau)} \tag{B.10d}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \mathtt{all}(t.\tau) \qquad \Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau}' \rightsquigarrow \tau' \ \mathsf{type}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prtap\{\grave{\tau}'\}}(\grave{e}) \rightsquigarrow \mathtt{tap\{\tau'\}}(e) : [\tau'/t]\tau} \tag{B.10e}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau} \rightsquigarrow \tau \ \mathsf{type} \qquad \Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : [\mathtt{rec}(t.\tau)/t]\tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prfold\{}t.\grave{\tau}\mathtt{\}}(\grave{e}) \rightsquigarrow \mathtt{fold\{}t.\tau\mathtt{\}}(e) : \mathtt{rec}(t.\tau)} \tag{B.10f}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \mathtt{rec}(t.\tau)}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prunfold}(\grave{e}) \rightsquigarrow \mathtt{unfold}(e) : [\mathtt{rec}(t.\tau)/t]\tau} \tag{B.10g}$$

$$\frac{\{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e}_i \rightsquigarrow e_i : \tau_i\}_{i \in L}}{\left(\begin{array}{c}\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prtpl\{}L\mathtt{\}}(\{i \hookrightarrow \grave{e}_i\}_{i \in L}) \\ \rightsquigarrow \\ \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) : \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})\end{array}\right)} \tag{B.10h}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \mathtt{prod}[L,\ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prprj}[\ell](\grave{e}) \rightsquigarrow \mathtt{prj}[\ell](e) : \tau} \tag{B.10i}$$

$$\frac{\{\Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau}_i \rightsquigarrow \tau_i \ \mathsf{type}\}_{i \in L} \qquad \Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau} \rightsquigarrow \tau \ \mathsf{type} \qquad \Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau}{\left(\begin{array}{c}\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prinj}[L,\ell;\ell]\{\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}; \ell \hookrightarrow \grave{\tau}\}(\grave{e}) \\ \rightsquigarrow \\ \mathtt{inj}[L,\ell;\ell]\{\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau\}(e) : \mathtt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)\end{array}\right)} \tag{B.10j}$$

$$\frac{\begin{array}{c}\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \\ \Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau} \rightsquigarrow \tau \ \mathsf{type} \qquad \{\Delta\ \Gamma, x_i : \tau_i \vdash^{\mathbb{E}} \grave{e}_i \rightsquigarrow e_i : \tau\}_{i \in L}\end{array}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prcase}[L]\{\grave{\tau}\}(\grave{e}; \{i \hookrightarrow x_i.\grave{e}_i\}_{i \in L}) \rightsquigarrow \mathtt{case}[L]\{\tau\}(e; \{i \hookrightarrow x_i.e_i\}_{i \in L}) : \tau} \tag{B.10k}$$

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

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau \qquad \Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau}' \rightsquigarrow \tau' \ \mathsf{type} \qquad \{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r}_i \rightsquigarrow r_i : \tau \Rightarrow \tau'\}_{1 \le i \le n}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prmatch}[n]\{\grave{\tau}'\}(\grave{e}; \{\grave{r}_i\}_{1 \le i \le n}) \rightsquigarrow \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \le i \le n}) : \tau'} \tag{B.10m}$$

$\boxed{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r} \rightsquigarrow r : \tau \Rightarrow \tau'}$ $\grave{r}$ has expansion r taking values of type r to values of type τ'

$$\frac{\Delta \vdash p : \tau \dashv \Gamma \qquad \Delta\ \Gamma \cup \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{prrule}(p.\grave{e}) \rightsquigarrow \mathtt{rule}(p.e) : \tau \Rightarrow \tau'} \tag{B.11}$$

B.3.4 Proto-Pattern Validation

Pattern splicing scenes, $\mathbb{P}$, are of the form $\Delta; \hat{\Phi}; b$.

$\boxed{\vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : \tau \dashv\!|^{\hat{\Gamma}}}$ $\grave{p}$ has expansion p matching against τ generating hypotheses $\hat{\Gamma}$

$$\frac{}{\vdash^{\mathbb{P}} \mathtt{prwildp} \rightsquigarrow \mathtt{wildp} : \tau \dashv\!|^{\langle \emptyset;\emptyset \rangle}} \tag{B.12a}$$

$$\frac{\vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : [\mathtt{rec}(t.\tau)/t]\tau \dashv\!|^{\hat{\Gamma}}}{\vdash^{\mathbb{P}} \mathtt{prfoldp}(\grave{p}) \rightsquigarrow \mathtt{foldp}(p) : \mathtt{rec}(t.\tau) \dashv\!|^{\hat{\Gamma}}} \tag{B.12b}$$

$$\frac{\{\vdash^{\mathbb{P}} \grave{p}_i \rightsquigarrow p_i : \tau_i \dashv\!|^{\hat{\Gamma}_i}\}_{i \in L}}{\left(\begin{array}{c} \vdash^{\mathbb{P}} \mathtt{prtplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i \in L}) \\ \rightsquigarrow \\ \mathtt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\!|^{\cup_{i \in L} \hat{\Gamma}_i} \end{array}\right)} \tag{B.12c}$$

$$\frac{\vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : \tau \dashv\!|^{\hat{\Gamma}}}{\vdash^{\mathbb{P}} \mathtt{prinjp}[\ell](\grave{p}) \rightsquigarrow \mathtt{injp}[\ell](p) : \mathtt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\!|^{\hat{\Gamma}}} \tag{B.12d}$$

$$\frac{\mathsf{parseUPat}(\mathsf{subseq}(b; m; n)) = \hat{p} \qquad \Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!|\ \hat{\Gamma}}{\vdash^{\Delta;\hat{\Phi};b} \mathtt{splicedp}[m;n] \rightsquigarrow p : \tau \dashv\!|^{\hat{\Gamma}}} \tag{B.12e}$$

B.4 Metatheory

B.4.1 Type Expansion

Lemma B.26 (Type Expansion). *If* $\langle \mathcal{D}; \Delta \rangle \vdash \hat{\tau} \rightsquigarrow \tau$ type *then* $\Delta \vdash \tau$ 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.27 (Proto-Type Validation). *If* $\Delta \vdash^{\langle \mathcal{D}; \Delta_{app} \rangle; b} \dot{\tau} \rightsquigarrow \tau$ type *and* $\Delta \cap \Delta_{app} = \emptyset$ *then* $\Delta \cup \Delta_{app} \vdash \tau$ type.

Proof. By rule induction over Rules (B.9).

Case (B.9a). We have

(1)	$\Delta = \Delta', t$ type	by assumption
(2)	$\dot{\tau} = t$	by assumption
(3)	$\tau = t$	by assumption
(4)	Δ', t type $\vdash t$ type	by Rule (B.1a)
(5)	Δ', t type $\cup \Delta_{\text{app}} \vdash t$ type	by Lemma B.2 over Δ_{app} to (4)

Case (B.9b).

(1)	$\dot{\tau} = \texttt{prparr}(\dot{\tau}_1; \dot{\tau}_2)$	by assumption
(2)	$\tau = \texttt{parr}(\tau_1; \tau_2)$	by assumption
(3)	$\Delta \vdash^{\langle \mathcal{D}; \Delta_{\text{app}} \rangle; b} \dot{\tau}_1 \rightsquigarrow \tau_1$ type	by assumption
(4)	$\Delta \vdash^{\langle \mathcal{D}; \Delta_{\text{app}} \rangle; b} \dot{\tau}_2 \rightsquigarrow \tau_2$ type	by assumption
(5)	$\Delta \cup \Delta_{\text{app}} \vdash \tau_1$ type	by IH on (3)
(6)	$\Delta \cup \Delta_{\text{app}} \vdash \tau_2$ type	by IH on (4)
(7)	$\Delta \cup \Delta_{\text{app}} \vdash \texttt{parr}(\tau_1; \tau_2)$ type	by Rule (B.1b) on (5) and (6)

Case (B.9c).

(1)	$\dot{\tau} = \texttt{prall}(t.\dot{\tau}')$	by assumption
(2)	$\tau = \texttt{all}(t.\tau')$	by assumption
(3)	Δ, t type $\vdash^{\langle \mathcal{D}; \Delta_{\text{app}} \rangle; b} \dot{\tau}' \rightsquigarrow \tau'$ type	by assumption
(4)	Δ, t type $\cup \Delta_{\text{app}} \vdash \tau'$ type	by IH on (3)
(5)	$\Delta \cup \Delta_{\text{app}}, t$ type $\vdash \tau'$ type	by exchange over Δ_{app} on (4)
(6)	$\Delta \cup \Delta_{\text{app}} \vdash \texttt{all}(t.\tau')$ type	by Rule (B.1c) on (5)

Case (B.9d).

(1)	$\dot{\tau} = \texttt{prrec}(t.\dot{\tau}')$	by assumption
(2)	$\tau = \texttt{rec}(t.\tau')$	by assumption
(3)	Δ, t type $\vdash^{\Delta_{\text{app}}; b} \dot{\tau}' \rightsquigarrow \tau'$ type	by assumption
(4)	Δ, t type $\cup \Delta_{\text{app}} \vdash \tau'$ type	by IH on (3)
(5)	$\Delta \cup \Delta_{\text{app}}, t$ type $\vdash \tau'$ type	by exchange over Δ_{app} on (4)
(6)	$\Delta \cup \Delta_{\text{app}} \vdash \texttt{rec}(t.\tau')$ type	by Rule (B.1d) on (5)

Case (B.9e).

(1)	$\dot{\tau} = \texttt{prprod}[L](\{i \hookrightarrow \dot{\tau}_i\}_{i \in L})$	by assumption
(2)	$\tau = \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})$	by assumption
(3)	$\{\Delta \vdash^{\Delta_{\text{app}}; b} \dot{\tau}_i \rightsquigarrow \tau_i$ type$\}_{i \in L}$	by assumption

(4) $\{\Delta \cup \Delta_{\text{app}} \vdash \tau_i \text{ type}\}_{i \in L}$ by IH over (3)
(5) $\Delta \cup \Delta_{\text{app}} \vdash \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})$ type by Rule (B.1e) on (4)

Case (B.9f).
(1) $\grave{\tau} = \mathtt{prsum}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L})$ by assumption
(2) $\tau = \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L})$ by assumption
(3) $\{\Delta \vdash^{\Delta_{\text{app}};b} \grave{\tau}_i \rightsquigarrow \tau_i \text{ type}\}_{i \in L}$ by assumption
(4) $\{\Delta \cup \Delta_{\text{app}} \vdash \tau_i \text{ type}\}_{i \in L}$ by IH over (3)
(5) $\Delta \cup \Delta_{\text{app}} \vdash \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L})$ type by Rule (B.1f) on (4)

Case (B.9g).
(1) $\grave{\tau} = \mathtt{splicedt}[m;n]$ by assumption
(2) $\mathsf{parseUTyp}(\mathsf{subseq}(b;m;n)) = \hat{\tau}$ by assumption
(3) $\langle \mathcal{D}; \Delta_{\text{app}} \rangle \vdash \hat{\tau} \rightsquigarrow \tau$ type by assumption
(4) $\Delta \cap \Delta_{\text{app}} = \emptyset$ by assumption
(5) $\Delta_{\text{app}} \vdash \tau$ type by Lemma B.26 on (3)
(6) $\Delta \cup \Delta_{\text{app}} \vdash \tau$ type by Lemma B.2 over Δ on (5) and exchange over Δ

□

B.4.2 Typed Pattern Expansion

Theorem B.28 (Typed Pattern Expansion). *Both of the following hold:*
1. *If* $\Delta \vdash_{\langle \mathcal{A};\Phi \rangle} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \langle \mathcal{G};\Gamma \rangle$ *then* $\Delta \vdash p : \tau \dashv\!\mid \Gamma$.
2. *If* $\vdash^{\Delta;\langle \mathcal{A};\Phi \rangle;b} \grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\langle \mathcal{G};\Gamma \rangle}$ *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 $\hat{\Gamma} = \langle \mathcal{G};\Gamma \rangle$ 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) $p = \mathtt{wildp}$ by assumption
(2) $\Gamma = \emptyset$ by assumption
(3) $\Delta \vdash \mathtt{wildp} : \tau \dashv\!\mid \emptyset$ by Rule (B.4b)
Case (B.8c).
(1) $\hat{p} = \mathtt{fold}(\hat{p}')$ by assumption
(2) $p = \mathtt{foldp}(p')$ by assumption

(3) $\tau = \texttt{rec}(t.\tau')$ by assumption
(4) $\Delta \vdash_{\hat{\Phi}} \hat{p}' \rightsquigarrow p' : [\texttt{rec}(t.\tau')/t]\tau' \dashv\Vdash \hat{\Gamma}$ by assumption
(5) $\Delta \vdash p' : [\texttt{rec}(t.\tau')/t]\tau' \dashv\Vdash \Gamma$ by IH, part 1 on (4)
(6) $\Delta \vdash \texttt{foldp}(p') : \texttt{rec}(t.\tau') \dashv\Vdash \Gamma$ 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) $\{\Delta \vdash_{\hat{\Phi}} \hat{p}_i \rightsquigarrow p_i : \tau_i \dashv\Vdash \langle \mathcal{G}_i; \Gamma_i \rangle\}_{i \in L}$ by assumption
(5) $\Gamma = \cup_{i \in L} \Gamma_i$ by assumption
(6) $\{\Delta \vdash p_i : \tau_i \dashv\Vdash \Gamma_i\}_{i \in L}$ by IH, part 1 over (4)
(7) $\Delta \vdash \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\Vdash \cup_{i \in L} \Gamma_i$ 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) $\Delta \vdash_{\hat{\Phi}} \hat{p}' \rightsquigarrow p' : \tau' \dashv\Vdash \hat{\Gamma}$ by assumption
(5) $\Delta \vdash p' : \tau' \dashv\Vdash \Gamma$ by IH, part 1 on (4)
(6) $\Delta \vdash \texttt{injp}[\ell](p') : \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau') \dashv\Vdash \Gamma$ 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) $b \downarrow_{\text{Body}} e_{\text{body}}$ by assumption
(5) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{SuccessP} \cdot e_{\text{proto}}$ by assumption
(6) $e_{\text{proto}} \uparrow_{\text{ProtoPat}} \hat{p}$ by assumption
(7) $\vdash^{\Delta; \langle \mathcal{A}; \Phi \rangle; b} \hat{p} \rightsquigarrow p : \tau \dashv\Vdash^{\langle \mathcal{G}; \Gamma \rangle}$ by assumption
(8) $\Delta \vdash p : \tau \dashv\Vdash \Gamma$ by IH, part 2 on (7)

2. We induct on the premise. In the following, let $\hat{\Gamma} = \langle \mathcal{G}; \Gamma \rangle$ and $\hat{\Phi} = \langle \mathcal{A}; \Phi \rangle$.

Case (B.12a).

(1) $p = \texttt{wildp}$ by assumption
(2) $\Gamma = \emptyset$ by assumption
(3) $\Delta \vdash \texttt{wildp} : \tau \dashv\Vdash \emptyset$ by Rule (B.4b)

Case (B.12b).

(1) $\hat{p} = \texttt{prfoldp}(\hat{p}')$ by assumption
(2) $p = \texttt{foldp}(p')$ by assumption
(3) $\tau = \texttt{rec}(t.\tau')$ by assumption

(4) $\vdash^{\Delta;\hat{\Phi};b} \grave{p}' \rightsquigarrow p' : [\mathtt{rec}(t.\tau')/t]\tau' \dashv\!\mid^{\hat{\Gamma}}$ 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) $\{\vdash^{\Delta;\hat{\Phi};b} \grave{p}_i \rightsquigarrow p_i : \tau_i \dashv\!\mid^{\langle \mathcal{G}_i;\Gamma_i\rangle}\}_{i\in L}$ 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 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) $\vdash^{\Delta;\hat{\Phi};b} \grave{p}' \rightsquigarrow p' : \tau' \dashv\!\mid^{\hat{\Gamma}}$ 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) $\grave{p} = \mathtt{splicedp}[m;n]$ by assumption
(2) $\mathsf{parseUExp}(\mathsf{subseq}(b;m;n)) = \hat{p}$ by assumption
(3) $\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \hat{\Gamma}$ by assumption
(4) $\Delta \vdash p : \tau \dashv\!\mid \Gamma$ by IH, part 1 on (3)

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:

$$\|\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \hat{\Gamma}\| = \|\hat{p}\|$$
$$\|\vdash^{\Delta;\hat{\Phi};b} \grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\hat{\Gamma}}\| = \|b\|$$

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:

$$\begin{aligned}&\|\Delta \vdash_{\hat{\Phi}} \hat{a}\ /b/ \rightsquigarrow p : \tau \dashv\!\mid \hat{\Gamma}\|\\ =&\|\vdash^{\Delta;\hat{\Phi};b} \grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\hat{\Gamma}}\|\\ =&\|b\|\end{aligned}$$

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 $\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \hat{\Gamma}$. 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:

$$\|\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \hat{\Gamma}\| < \|\vdash^{\Delta;\hat{\Phi};b} \mathtt{splicedp}[m;n] \rightsquigarrow p : \tau \dashv\!\mid^{\hat{\Gamma}}\|$$

i.e. by the definitions above,

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

This is established by appeal to Condition B.22, which states that subsequences of b are no longer than b, and the Condition B.14, 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.22 and B.14, we have that $\|\hat{p}\| < \|b\|$ as needed. □

B.4.3 Typed Expression Expansion

Theorem B.29 (Typed Expansion (Full))**.** *All of the following hold:*

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 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)	$\hat{e} = \lambda \hat{x}{:}\hat{\tau}_1.\hat{e}'$	by assumption
(2)	$e = \mathtt{lam\{}\tau_1\mathtt{\}(}x.e'\mathtt{)}$	by assumption
(3)	$\tau = \mathtt{parr(}\tau_1;\tau_2\mathtt{)}$	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.26 on (4)
(7)	$\Delta\ \Gamma, x : \tau_1 \vdash e' : \tau_2$	by IH, part 1(a) on (5)

(8) $\Delta\ \Gamma \vdash \texttt{lam\{}\tau_1\texttt{\}(}x.e'\texttt{)} : \texttt{parr(}\tau_1\texttt{;}\tau_2\texttt{)}$ by Rule (B.2b) on (6) and (7)

Case (B.6c).

(1) $\hat{e} = \hat{e}_1(\hat{e}_2)$ by assumption
(2) $e = \texttt{ap(}e_1\texttt{;}e_2\texttt{)}$ by assumption
(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_1 \rightsquigarrow e_1 : \texttt{parr(}\tau_2\texttt{;}\tau\texttt{)}$ by assumption
(4) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_2 \rightsquigarrow e_2 : \tau_2$ by assumption
(5) $\Delta\ \Gamma \vdash e_1 : \texttt{parr(}\tau_2\texttt{;}\tau\texttt{)}$ by IH, part 1(a) on (3)
(6) $\Delta\ \Gamma \vdash e_2 : \tau_2$ by IH, part 1(a) on (4)
(7) $\Delta\ \Gamma \vdash \texttt{ap(}e_1\texttt{;}e_2\texttt{)} : \tau$ by Rule (B.2c) on (5) and (6)

Case (B.6d) **through** (B.6k). These cases follow analagously, i.e. we apply Lemma B.26 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.6l). We have

(1) $\hat{e} = \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}'\ \texttt{by}\ \texttt{static}\ e_{\text{parse}}\ \texttt{end}\ \texttt{in}\ \hat{e}'$ by assumption
(2) $\hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'\ \text{type}$ 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) $\Delta \vdash \tau'\ \text{type}$ by Lemma B.26 to (2)
(6) $\Delta\ \Gamma \vdash e : \tau$ by IH, part 1(a) on (4)

Case (B.6m). We have

(1) $\hat{e} = \hat{a}\ /b/$ 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) $b \downarrow_{\text{Body}} e_{\text{body}}$ by assumption
(5) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{SuccessE} \cdot e_{\text{proto}}$ by assumption
(6) $e_{\text{proto}} \uparrow_{\text{ProtoExpr}} \grave{e}$ 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.6n).

(1) $\hat{e} = \texttt{match}\ \hat{e}'\ \{\hat{r}_i\}_{1 \le i \le n}$ — by assumption
(2) $e = \texttt{match}[n]\{\tau\}(e'; \{r_i\}_{1 \le i \le n})$ — by assumption
(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \rightsquigarrow e' : \tau'$ — by assumption
(4) $\Delta \vdash \tau\ \text{type}$ — by assumption
(5) $\{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \rightsquigarrow r_i : \tau' \Mapsto \tau\}_{1 \le i \le n}$ — by assumption
(6) $\Delta\ \Gamma \vdash e' : \tau'$ — by IH, part 1(a) on (3)
(7) $\{\Delta\ \Gamma \vdash r_i : \tau' \Mapsto \tau\}_{1 \le i \le n}$ — by IH, part 1(b) over (5)
(8) $\Delta\ \Gamma \vdash \texttt{match}[n]\{\tau\}(e'; \{r_i\}_{1 \le i \le n}) : \tau$ — by Rule (B.21) on (6), (4) and (7)

Case (B.6o).

(1) $\hat{e} = \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}'\ \texttt{for patterns by static}\ e_{\text{parse}}\ \texttt{end in}\ \hat{e}'$ — by assumption
(2) $\hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'\ \text{type}$ — 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) $\Delta \vdash \tau'\ \text{type}$ — by Lemma B.26 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) $\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \langle \mathcal{A}'; \Gamma \rangle$ — by assumption
(4) $\hat{\Delta}\ \langle \mathcal{A} \uplus \mathcal{A}'; \Gamma \cup \Gamma \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau'$ — by assumption
(5) $\Delta \vdash p : \tau \dashv\!\mid \Gamma$ — by Theorem B.28, 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.10b).

(1) $\grave{e} = \texttt{prlam\{}\grave{\tau}_1\texttt{\}(}x.\grave{e}'\texttt{)}$ by assumption
(2) $e = \texttt{lam\{}\tau_1\texttt{\}(}x.e'\texttt{)}$ by assumption
(3) $\tau = \texttt{parr(}\tau_1\texttt{;}\tau_2\texttt{)}$ 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.27 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) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash \texttt{lam\{}\tau_1\texttt{\}(}x.e'\texttt{)} : \texttt{parr(}\tau_1\texttt{;}\tau_2\texttt{)}$ by Rule (B.2b) on (10) and (12)

Case (B.10c).

(1) $\grave{e} = \texttt{prap(}\grave{e}_1\texttt{;}\grave{e}_2\texttt{)}$ by assumption
(2) $e = \texttt{ap(}e_1\texttt{;}e_2\texttt{)}$ by assumption
(3) $\Delta\ \Gamma \vdash^{\hat{\Delta}_{\text{app}};\hat{\Gamma}_{\text{app}};\hat{\Psi};\hat{\Phi};b} \grave{e}_1 \rightsquigarrow e_1 : \texttt{parr(}\tau_2\texttt{;}\tau\texttt{)}$ by assumption
(4) $\Delta\ \Gamma \vdash^{\hat{\Delta}_{\text{app}};\hat{\Gamma}_{\text{app}};\hat{\Psi};\hat{\Phi};b} \grave{e}_2 \rightsquigarrow e_2 : \tau_2$ 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) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash e_1 : \texttt{parr(}\tau_2\texttt{;}\tau\texttt{)}$ 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) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash \texttt{ap(}e_1\texttt{;}e_2\texttt{)} : \tau$ by Rule (B.2c) on (7) and (8)

Case (B.10d).

(1) $\grave{e} = \texttt{prtlam(}t.\grave{e}'\texttt{)}$ by assumption
(2) $e = \texttt{tlam(}t.e'\texttt{)}$ by assumption
(3) $\tau = \texttt{all(}t.\tau'\texttt{)}$ by assumption
(4) Δ, t type $\Gamma \vdash^{\hat{\Delta}_{\text{app}};\hat{\Gamma}_{\text{app}};\hat{\Psi};\hat{\Phi};b} \grave{e}' \rightsquigarrow e' : \tau'$ by assumption
(5) $\Delta \cap \Delta_{\text{app}} = \emptyset$ by assumption
(6) $\text{dom}(\Gamma) \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset$ by assumption

(7) t type $\notin \Delta_{\text{app}}$ — by identification convention

(8) Δ, t type $\cap\, \Delta_{\text{app}} = \emptyset$ — by (5) and (7)

(9) Δ, t type $\cup\, \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash e' : \tau'$ — by IH, part 2(a) on (4), (8) and (6)

(10) $\Delta \cup \Delta_{\text{app}}, t$ 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.10e) **through** (B.10k). These cases follow analagously, i.e. we apply the IH, part 2(a) to all proto-expression validation judgements, Lemma B.27 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.10l).

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

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

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

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

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

(6) $\Delta_{\text{app}}\ \Gamma_{\text{app}} \vdash e : \tau$ — by IH, part 1(a) on (3)

(7) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash e : \tau$ — by Lemma B.2 over Δ and Γ and exchange on (6)

Case (B.10m).

(1) $\grave{e} = \texttt{prmatch}[n]\{\grave{\tau}\}(\grave{e}'; \{\grave{r}_i\}_{1 \leq i \leq n})$ — by assumption

(2) $e = \texttt{match}[n]\{\tau\}(e'; \{r_i\}_{1 \leq i \leq n})$ — by assumption

(3) $\Delta\ \Gamma \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e}' \rightsquigarrow e' : \tau'$ — by assumption

(4) $\Delta \vdash^{\hat{\Delta};b} \grave{\tau} \rightsquigarrow \tau$ type — by assumption

(5) $\{\Delta\ \Gamma \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{r}_i \rightsquigarrow r_i : \tau' \Rightarrow \tau\}_{1 \leq i \leq n}$ — 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) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash e' : \tau'$ — by IH, part 2(a) on (3), (6) and (7)

(9) $\Delta \cup \Delta_{\text{app}} \vdash \tau$ type — by Lemma B.27 on (4)

(10) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash r : \tau' \Rightarrow \tau$ — by IH, part 2(b) on (5), (6) and (7)

(11) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash \texttt{match}[n]\{\tau\}(e'; \{r_i\}_{1 \leq i \leq n}) : \tau$

	by Rule (B.21) on (8), (9), (10)

(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) $\Delta \vdash p : \tau \dashv\Vdash \Gamma$	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) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash \texttt{rule}(p.e) : \tau \Rightarrowtriangle \tau'$	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.6m). There, we have that the metric remains stable:

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

The only case in the proof of part 2 that invokes part 1 is Case (B.10l). There, we have that $\text{parseUExp}(\text{subseq}(b; m; n)) = \hat{e}$ 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:

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

i.e. by the definitions above,

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

This is established by appeal to Condition B.22, which states that subsequences of b are no longer than b, and Condition B.13, 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.30 (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.29, part 1(a). □

B.4.4 Reasoning Principles

The following theorem, together with Theorem B.30, establishes **Typing**, **Segmentation** and **Context Independence** as discussed in Sec. 3.1.4.

Theorem B.31 (Typing, Segmentation and Context Independence). *If* $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{a}\ /b/ \rightsquigarrow e : \tau$ *then:*

1. *(**Typing**)* $\hat{\Psi} = \hat{\Psi}', \hat{a} \rightsquigarrow a \hookrightarrow \texttt{setsm}(\tau; e_{parse})$
2. $b \downarrow_{\mathsf{Body}} e_{body}$
3. $e_{parse}(e_{body}) \Downarrow \texttt{SuccessE} \cdot e_{proto}$
4. $e_{proto} \uparrow_{\mathsf{ProtoExpr}} \grave{e}$
5. *(**Segmentation**)* $\mathsf{seg}(\grave{e})$ segments b
6. *(**Context Independence**)* $\emptyset\ \emptyset \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e : \tau$

Proof. By rule induction over Rules (B.6). The only rule that applies is Rule (B.6m). The conclusions of the theorem are the premises of this rule. □

The following theorem establishes a prohibition on **Shadowing** as discussed in Sec. 3.1.4.

Theorem B.32 (Shadowing Prohibition).

1. *If* $\Delta \vdash^{\langle \mathcal{D};\Delta_{app}\rangle;b} \texttt{splicedt}[m;n] \rightsquigarrow \tau$ type *then:*
   (a) $\mathsf{parseUTyp}(\mathsf{subseq}(b;m;n)) = \hat{\tau}$
   (b) $\langle \mathcal{D}; \Delta_{app} \rangle \vdash \hat{\tau} \rightsquigarrow \tau$ type
   (c) $\Delta \cap \Delta_{app} = \emptyset$
2. *If* $\Delta\ \Gamma \vdash^{\langle \mathcal{D};\Delta_{app}\rangle;\langle \mathcal{G};\Gamma_{app}\rangle;\hat{\Psi};\hat{\Phi};b} \texttt{splicede}[m;n] \rightsquigarrow e : \tau$ *then:*
   (a) $\mathsf{parseUExp}(\mathsf{subseq}(b;m;n)) = \hat{e}$
   (b) $\langle \mathcal{D}; \Delta_{app} \rangle\ \langle \mathcal{G}; \Gamma_{app} \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau$

(c) $\Delta \cap \Delta_{app} = \emptyset$
(d) $dom(\Gamma) \cap dom(\Gamma_{app}) = \emptyset$

Proof.

1. By rule induction over Rules (B.9). The only rule that applies is Rule (B.9g). The conclusions are the premises of tihs rule.
2. By rule induction over Rules (B.10). The only rule that applies is Rule (B.10l). The conclusions are the premises of tihs rule.

□

The following theorem, together with Theorem B.28 part 1, establishes **Typing** and **Segmentation**, as discussed in Sec. 4.1.4.

Theorem B.33 (spTSM Typing and Segmentation)**.** *If* $\Delta \vdash_{\hat{\Phi}} \hat{a}\ /b/ \rightsquigarrow p : \tau \dashv\mid \hat{\Gamma}$ *then*

1. *(**Typing**)* $\hat{\Phi} = \hat{\Phi}', \hat{a} \rightsquigarrow a \hookrightarrow \texttt{sptsm}(\tau; e_{parse})$
2. $b \downarrow_{\mathsf{Body}} e_{body}$
3. $e_{parse}(e_{body}) \Downarrow \texttt{SuccessP} \cdot e_{proto}$
4. $e_{proto} \uparrow_{\mathsf{ProtoPat}} \grave{p}$
5. *(**Segmentation**)* $\mathsf{seg}(\grave{p})$ segments b

Proof. By rule induction over Rules (B.8). The only rule that applies is Rule (B.8f). The conclusions are premises of this rule. □

Appendix C

miniVerse$_{\mathbf{P}}$

TODO: add static environments and introductory explanation

C.1 Expanded Language (XL)

C.1.1 Syntax

Signatures and Module Expressions

Sort			Operational Form	Description
Sig	σ	$::=$	$\texttt{sig}\{\kappa\}(u.\tau)$	signature
Mod	M	$::=$	X	variable
			$\texttt{struct}(c;e)$	structure
			$\texttt{seal}\{\sigma\}(M)$	seal
			$\texttt{mlet}\{\sigma\}(M;X.M)$	definition

Kinds and Constructors

Sort			Operational Form	Description
Kind	κ	$::=$	$\texttt{darr}(\kappa;u.\kappa)$	dependent function
			$\texttt{unit}$	nullary product
			$\texttt{dprod}(\kappa;u.\kappa)$	dependent product
			$\texttt{Type}$	types
			$\texttt{S}(\tau)$	singleton
Con	c,τ	$::=$	u	variable
			t	
			$\texttt{abs}(u.c)$	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
			$\texttt{con}(M)$	constructor part

Expressions, Rules and Patterns

Sort			Operational Form	Description
Exp	e	::=	x	variable
			$\texttt{lam}\{\tau\}(x.e)$	abstraction
			$\texttt{ap}(e;e)$	application
			$\texttt{clam}\{\kappa\}(u.e)$	constructor abstraction
			$\texttt{cap}\{\kappa\}(e)$	constructor application
			$\texttt{fold}\{t.\tau\}(e)$	fold
			$\texttt{unfold}(e)$	unfold
			$\texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L})$	labeled tuple
			$\texttt{prj}[\ell](e)$	projection
			$\texttt{inj}[L;\ell]\{\{i \hookrightarrow \tau_i\}_{i\in L}\}(e)$	injection
			$\texttt{match}[n]\{\tau\}(e;\{r_i\}_{1\le i\le n})$	match
			$\texttt{val}(M)$	value part
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

- $\Omega, x : \tau$ when $x \notin \mathrm{dom}(\Omega)$ and $\Omega \vdash \tau :: \texttt{Type}$ for the extension of Ω with a mapping from x to the hypothesis $x : \tau$
- $\Omega, u :: \kappa$ when $u \notin \mathrm{dom}(\Omega)$ and $\Omega \vdash \kappa\ \mathsf{kind}$ for the extension of Ω with a mapping from u to the hypothesis $u :: \kappa$
- $\Omega, X : \sigma$ when $X \notin \mathrm{dom}(\Omega)$ and $\Omega \vdash \sigma\ \mathsf{sig}$ for the extension of Ω with a mapping from X to the hypothesis $X : \sigma$.

A well-formed unified context is one that can be constructed by some sequence of such extensions, start from the empty context, $\emptyset$. We identify unified contexts up to exchange and contraction in the usual manner.

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

$$\frac{\Omega \vdash \kappa \equiv \kappa' \qquad \Omega, u :: \kappa \vdash \tau \equiv \tau' \ \mathsf{type}}{\Omega \vdash \mathtt{sig\{}\kappa\mathtt{\}(}u.\tau\mathtt{)} \equiv \mathtt{sig\{}\kappa'\mathtt{\}(}u.\tau'\mathtt{)}} \tag{C.2}$$

$\boxed{\Omega \vdash \sigma <: \sigma'}$ σ is a subsignature of σ'

$$\frac{\Omega \vdash \kappa <:: \kappa' \qquad \Omega, u :: \kappa \vdash \tau <: \tau'}{\Omega \vdash \mathtt{sig\{}\kappa\mathtt{\}(}u.\tau\mathtt{)} <: \mathtt{sig\{}\kappa'\mathtt{\}(}u.\tau'\mathtt{)}} \tag{C.3}$$

$\boxed{\Omega \vdash M : \sigma}$ M matches σ

$$\frac{\Omega \vdash M : \sigma \qquad \Omega \vdash \sigma <: \sigma'}{\Omega \vdash M : \sigma'} \tag{C.4a}$$

$$\frac{}{\Omega, X : \sigma \vdash X : \sigma} \tag{C.4b}$$

$$\frac{\Omega \vdash c :: \kappa \qquad \Omega \vdash e : [c/u]\tau}{\Omega \vdash \mathtt{struct(}c; e\mathtt{)} : \mathtt{sig\{}\kappa\mathtt{\}(}u.\tau\mathtt{)}} \tag{C.4c}$$

$$\frac{\Omega \vdash \sigma \ \mathsf{sig} \qquad \Omega \vdash M : \sigma}{\Omega \vdash \mathtt{seal\{}\sigma\mathtt{\}(}M\mathtt{)} : \sigma} \tag{C.4d}$$

$$\frac{\Omega \vdash M : \sigma \qquad \Omega \vdash \sigma' \ \mathsf{sig} \qquad \Omega, X : \sigma \vdash M' : \sigma'}{\Omega \vdash \mathtt{mlet\{}\sigma'\mathtt{\}(}M; X.M'\mathtt{)} : \sigma'} \tag{C.4e}$$

$\boxed{\Omega \vdash M \ \mathsf{mval}}$ M is, or stands for, a module value

$$\frac{}{\Omega \vdash \mathtt{struct(}c; e\mathtt{)} \ \mathsf{mval}} \tag{C.5a}$$

$$\frac{}{\Omega, X : \sigma \vdash X \ \mathsf{mval}} \tag{C.5b}$$

Kinds and Constructors

$\boxed{\Omega \vdash \kappa \ \mathsf{kind}}$ κ is a kind

$$\frac{\Omega \vdash \kappa_1 \ \mathsf{kind} \qquad \Omega, u :: \kappa_1 \vdash \kappa_2 \ \mathsf{kind}}{\Omega \vdash \mathtt{darr(}\kappa_1; u.\kappa_2\mathtt{)} \ \mathsf{kind}} \tag{C.6a}$$

$$\frac{}{\Omega \vdash \mathtt{unit} \ \mathsf{kind}} \tag{C.6b}$$

$$\frac{\Omega \vdash \kappa_1 \text{ kind} \qquad \Omega, u :: \kappa_1 \vdash \kappa_2 \text{ kind}}{\Omega \vdash \texttt{dprod}(\kappa_1; u.\kappa_2) \text{ kind}} \tag{C.6c}$$

$$\frac{}{\Omega \vdash \texttt{Type kind}} \tag{C.6d}$$

$$\frac{\Omega \vdash \tau :: \texttt{Type}}{\Omega \vdash \texttt{S}(\tau) \text{ kind}} \tag{C.6e}$$

$\boxed{\Omega \vdash \kappa \equiv \kappa'}$ κ and κ' are definitionally equal

$$\frac{\Omega \vdash \kappa \text{ kind}}{\Omega \vdash \kappa \equiv \kappa} \tag{C.7a}$$

$$\frac{\Omega \vdash \kappa \equiv \kappa'}{\Omega \vdash \kappa' \equiv \kappa} \tag{C.7b}$$

$$\frac{\Omega \vdash \kappa \equiv \kappa' \qquad \Omega \vdash \kappa' \equiv \kappa''}{\Omega \vdash \kappa \equiv \kappa''} \tag{C.7c}$$

$$\frac{\Omega \vdash \kappa_1 \equiv \kappa_1' \qquad \Omega, u :: \kappa_1 \vdash \kappa_2 \equiv \kappa_2'}{\Omega \vdash \texttt{darr}(\kappa_1; u.\kappa_2) \equiv \texttt{darr}(\kappa_1'; u.\kappa_2')} \tag{C.7d}$$

$$\frac{\Omega \vdash \kappa_1 \equiv \kappa_1' \qquad \Omega, u :: \kappa_1 \vdash \kappa_2 \equiv \kappa_2'}{\Omega \vdash \texttt{dprod}(\kappa_1; u.\kappa_2) \equiv \texttt{dprod}(\kappa_1'; u.\kappa_2')} \tag{C.7e}$$

$$\frac{\Omega \vdash c \equiv c' :: \texttt{Type}}{\Omega \vdash \texttt{S}(c) \equiv \texttt{S}(c')} \tag{C.7f}$$

$\boxed{\Omega \vdash \kappa <:: \kappa'}$ κ is a subkind of κ'

$$\frac{\Omega \vdash \kappa \equiv \kappa'}{\Omega \vdash \kappa <:: \kappa'} \tag{C.8a}$$

$$\frac{\Omega \vdash \kappa <:: \kappa' \qquad \Omega \vdash \kappa' <:: \kappa''}{\Omega \vdash \kappa <:: \kappa''} \tag{C.8b}$$

$$\frac{\Omega \vdash \kappa_1' <:: \kappa_1 \qquad \Omega, u :: \kappa_1' \vdash \kappa_2 <:: \kappa_2'}{\Omega \vdash \texttt{darr}(\kappa_1; u.\kappa_2) <:: \texttt{darr}(\kappa_1'; u.\kappa_2')} \tag{C.8c}$$

$$\frac{\Omega \vdash \kappa_1 <:: \kappa_1' \qquad \Omega, u :: \kappa_1 \vdash \kappa_2 <:: \kappa_2'}{\Omega \vdash \texttt{dprod}(\kappa_1; u.\kappa_2) <:: \texttt{dprod}(\kappa_1'; u.\kappa_2')} \tag{C.8d}$$

$$\frac{\Omega \vdash \tau :: \texttt{Type}}{\Omega \vdash \texttt{S}(\tau) <:: \texttt{Type}} \tag{C.8e}$$

$$\frac{\Omega \vdash \tau <: \tau'}{\Omega \vdash \mathtt{S}(\tau) <:: \mathtt{S}(\tau')\mathtt{Type}} \tag{C.8f}$$

$\boxed{\Omega \vdash c :: \kappa}$ c has kind κ

$$\frac{\Omega \vdash c :: \kappa_1 \qquad \Omega \vdash \kappa_1 <:: \kappa_2}{\Omega \vdash c :: \kappa_2} \tag{C.9a}$$

$$\frac{}{\Omega, u :: \kappa \vdash u :: \kappa} \tag{C.9b}$$

$$\frac{\Omega, u :: \kappa_1 \vdash c_2 :: \kappa_2}{\Omega \vdash \mathtt{abs}(u.c_2) :: \mathtt{darr}(\kappa_1; u.\kappa_2)} \tag{C.9c}$$

$$\frac{\Omega \vdash c_1 :: \mathtt{darr}(\kappa_2; u.\kappa) \qquad \Omega \vdash c_2 :: \kappa_2}{\Omega \vdash \mathtt{app}(c_1; c_2) :: [c_1/u]\kappa} \tag{C.9d}$$

$$\frac{}{\Omega \vdash \mathtt{triv} :: \mathtt{unit}} \tag{C.9e}$$

$$\frac{\Omega \vdash c_1 :: \kappa_1 \qquad \Omega \vdash c_2 :: [c_1/u]\kappa_2}{\Omega \vdash \mathtt{pair}(c_1; c_2) :: \mathtt{dprod}(\kappa_1; u.\kappa_2)} \tag{C.9f}$$

$$\frac{\Omega \vdash c :: \mathtt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash \mathtt{prl}(c) :: \kappa_1} \tag{C.9g}$$

$$\frac{\Omega \vdash c :: \mathtt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash \mathtt{prr}(c) :: [\mathtt{prl}(c)/u]\kappa_2} \tag{C.9h}$$

$$\frac{\Omega \vdash \tau_1 :: \mathtt{Type} \qquad \Omega \vdash \tau_2 :: \mathtt{Type}}{\Omega \vdash \mathtt{parr}(\tau_1; \tau_2) :: \mathtt{Type}} \tag{C.9i}$$

$$\frac{\Omega \vdash \kappa \ \mathsf{kind} \qquad \Omega, u :: \kappa \vdash \tau :: \mathtt{Type}}{\Omega \vdash \mathtt{all}\{\kappa\}(u.\tau) :: \mathtt{Type}} \tag{C.9j}$$

$$\frac{\Omega, t :: \mathtt{Type} \vdash \tau :: \mathtt{Type}}{\Omega \vdash \mathtt{rec}(t.\tau) :: \mathtt{Type}} \tag{C.9k}$$

$$\frac{\{\Omega \vdash \tau_i :: \mathtt{Type}\}_{1 \le i \le n}}{\Omega \vdash \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) :: \mathtt{Type}} \tag{C.9l}$$

$$\frac{\{\Omega \vdash \tau_i :: \mathtt{Type}\}_{1 \le i \le n}}{\Omega \vdash \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) :: \mathtt{Type}} \tag{C.9m}$$

$$\frac{\Omega \vdash c :: \mathtt{Type}}{\Omega \vdash c :: \mathtt{S}(c)} \tag{C.9n}$$

$$\frac{\Omega \vdash M \ \mathsf{mval} \qquad \Omega \vdash M : \mathtt{sig}\{\kappa\}(u.\tau)}{\Omega \vdash \mathtt{con}(M) :: \kappa} \tag{C.9o}$$

$\boxed{\Omega \vdash c \equiv c' :: \kappa}$ c and c' are definitionally equal as constructors of kind κ

$$\frac{\Omega \vdash c :: \kappa}{\Omega \vdash c \equiv c :: \kappa} \tag{C.10a}$$

$$\frac{\Omega \vdash c \equiv c' :: \kappa}{\Omega \vdash c' \equiv c :: \kappa} \tag{C.10b}$$

$$\frac{\Omega \vdash c \equiv c' :: \kappa \qquad \Omega \vdash c' \equiv c'' :: \kappa}{\Omega \vdash c \equiv c'' :: \kappa} \tag{C.10c}$$

$$\frac{\Omega, u :: \kappa_1 \vdash c \equiv c' :: \kappa_2}{\Omega \vdash \mathtt{abs}(u.c) \equiv \mathtt{abs}(u.c') :: \mathtt{darr}(\kappa_1; u.\kappa_2)} \tag{C.10d}$$

$$\frac{\Omega \vdash c_1 \equiv c_1' :: \mathtt{darr}(\kappa_2; u.\kappa) \qquad \Omega \vdash c_2 \equiv c_2' :: \kappa_2}{\Omega \vdash \mathtt{app}(c_1; c_2) \equiv \mathtt{app}(c_1'; c_2') :: \kappa} \tag{C.10e}$$

$$\frac{\Omega \vdash \mathtt{abs}(u.c) :: \mathtt{darr}(\kappa_2; u.\kappa) \qquad \Omega \vdash c_2 :: \kappa_2}{\Omega \vdash \mathtt{app}(\mathtt{abs}(u.c); c_2) \equiv [c_2/u]c :: [c_2/u]\kappa} \tag{C.10f}$$

$$\frac{\Omega \vdash c_1 \equiv c_1' :: \kappa_1 \qquad \Omega \vdash c_2 \equiv c_2' :: [c_1/u]\kappa_2}{\Omega \vdash \mathtt{pair}(c_1; c_2) \equiv \mathtt{pair}(c_1'; c_2') :: \mathtt{dprod}(\kappa_1; u.\kappa_2)} \tag{C.10g}$$

$$\frac{\Omega \vdash c \equiv c' :: \mathtt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash \mathtt{prl}(c) \equiv \mathtt{prl}(c') :: \kappa_1} \tag{C.10h}$$

$$\frac{\Omega \vdash c_1 :: \kappa_1 \qquad \Omega \vdash c_2 :: \kappa_2}{\Omega \vdash \mathtt{prl}(\mathtt{pair}(c_1; c_2)) \equiv c_1 :: \kappa_1} \tag{C.10i}$$

$$\frac{\Omega \vdash c \equiv c' :: \mathtt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash \mathtt{prr}(c) \equiv \mathtt{prr}(c') :: [\mathtt{prl}(c)/u]\kappa_2} \tag{C.10j}$$

$$\frac{\Omega \vdash c_1 :: \kappa_1 \qquad \Omega \vdash c_2 :: \kappa_2}{\Omega \vdash \mathtt{prr}(\mathtt{pair}(c_1; c_2)) \equiv c_2 :: \kappa_2} \tag{C.10k}$$

$$\frac{\Omega \vdash \tau_1 \equiv \tau_1' :: \mathtt{Type} \qquad \Omega \vdash \tau_2 \equiv \tau_2' :: \mathtt{Type}}{\Omega \vdash \mathtt{parr}(\tau_1; \tau_2) \equiv \mathtt{parr}(\tau_1'; \tau_2') :: \mathtt{Type}} \tag{C.10l}$$

$$\frac{\Omega \vdash \kappa \equiv \kappa' \qquad \Omega, u :: \kappa \vdash \tau \equiv \tau' :: \mathtt{Type}}{\Omega \vdash \mathtt{all}\{\kappa\}(u.\tau) \equiv \mathtt{all}\{\kappa'\}(u.\tau') :: \mathtt{Type}} \tag{C.10m}$$

$$\frac{\Omega, t :: \mathtt{Type} \vdash \tau \equiv \tau' :: \mathtt{Type}}{\Omega \vdash \mathtt{rec}(t.\tau) \equiv \mathtt{rec}(t.\tau') :: \mathtt{Type}} \tag{C.10n}$$

$$\frac{\{\Omega \vdash \tau_i \equiv \tau_i' :: \mathtt{Type}\}_{1 \leq i \leq n}}{\Omega \vdash \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \equiv \mathtt{prod}[L](\{i \hookrightarrow \tau_i'\}_{i \in L}) :: \mathtt{Type}} \tag{C.10o}$$

$$\frac{\{\Omega \vdash \tau_i \equiv \tau_i' :: \mathtt{Type}\}_{1 \leq i \leq n}}{\Omega \vdash \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \equiv \mathtt{sum}[L](\{i \hookrightarrow \tau_i'\}_{i \in L}) :: \mathtt{Type}} \tag{C.10p}$$

$$\frac{\Omega \vdash c :: \mathtt{S}(c')}{\Omega \vdash c \equiv c' :: \mathtt{Type}} \tag{C.10q}$$

$$\frac{\Omega \vdash \mathtt{struct}(c; e) : \mathtt{sig}\{\kappa\}(u.\tau)}{\Omega \vdash \mathtt{con}(\mathtt{struct}(c; e)) \equiv c :: \kappa} \tag{C.10r}$$

Expressions, Rules and Patterns

$\boxed{\Omega \vdash \tau \ \mathsf{type}}$ τ is a type

Types, τ, classify expressions. The constructors of kind `Type` coincide with the types of miniVerse$_\mathbf{P}$.

$$\frac{\Omega \vdash \tau :: \mathtt{Type}}{\Omega \vdash \tau \ \mathsf{type}} \tag{C.11}$$

$\boxed{\Omega \vdash \tau \equiv \tau' \ \mathsf{type}}$ τ and τ' are definitionally equal types

Type equality then coincides with constructor equality at kind `Type`.

$$\frac{\Omega \vdash \tau \equiv \tau :: \mathtt{Type}}{\Omega \vdash \tau \equiv \tau' \ \mathsf{type}} \tag{C.12}$$

$\boxed{\Omega \vdash \tau <: \tau'}$ τ is a subtype of τ'

$$\frac{\Omega \vdash \tau_1 \equiv \tau_2 \ \mathsf{type}}{\Omega \vdash \tau_1 <: \tau_2} \tag{C.13a}$$

$$\frac{\Omega \vdash \tau <: \tau' \qquad \Omega \vdash \tau' <: \tau''}{\Omega \vdash \tau <: \tau''} \tag{C.13b}$$

$$\frac{\Omega \vdash \tau_1' <: \tau_1 \qquad \Omega \vdash \tau_2 <: \tau_2'}{\Omega \vdash \mathtt{parr}(\tau_1; \tau_2) <: \mathtt{parr}(\tau_1'; \tau_2')} \tag{C.13c}$$

$$\frac{\Omega \vdash \kappa' <:: \kappa \qquad \Omega, u :: \kappa \vdash \tau <: \tau'}{\Omega \vdash \mathtt{all}\{\kappa\}(u.\tau) <: \mathtt{all}\{\kappa'\}(u.\tau')} \tag{C.13d}$$

$$\frac{\{\Omega \vdash \tau_i <: \tau_i'\}_{i \in L}}{\Omega \vdash \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) <: \mathtt{prod}[L](\{i \hookrightarrow \tau_i'\}_{i \in L})} \tag{C.13e}$$

$$\frac{\{\Omega \vdash \tau_i <: \tau_i'\}_{i \in L}}{\Omega \vdash \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) <: \mathtt{sum}[L](\{i \hookrightarrow \tau_i'\}_{i \in L})} \tag{C.13f}$$

$\boxed{\Omega \vdash e : \tau}$ e has type τ

$$\frac{\Omega \vdash e : \tau \qquad \Omega \vdash \tau <: \tau'}{\Omega \vdash e : \tau'} \tag{C.14a}$$

$$\frac{}{\Omega, x : \tau \vdash x : \tau} \tag{C.14b}$$

$$\frac{\Omega \vdash \tau \ \mathsf{type} \qquad \Omega, x : \tau \vdash e : \tau'}{\Omega \vdash \mathtt{lam}\{\tau\}(x.e) : \mathtt{parr}(\tau; \tau')} \tag{C.14c}$$

$$\frac{\Omega \vdash e_1 : \mathtt{parr}(\tau; \tau') \qquad \Omega \vdash e_2 : \tau}{\Omega \vdash \mathtt{ap}(e_1; e_2) : \tau'} \tag{C.14d}$$

$$\frac{\Omega \vdash \kappa \ \mathsf{kind} \qquad \Omega, u :: \kappa \vdash e : \tau}{\Omega \vdash \mathtt{clam}\{\kappa\}(u.e) : \mathtt{all}\{\kappa\}(u.\tau)} \tag{C.14e}$$

$$\frac{\Omega \vdash e : \mathtt{all}\{\kappa\}(u.\tau) \qquad \Omega \vdash c :: \kappa}{\Omega \vdash \mathtt{cap}\{c\}(e) : [c/u]\tau} \tag{C.14f}$$

$$\frac{\Omega, t :: \mathsf{Type} \vdash \tau \ \mathsf{type} \qquad \Omega \vdash e : [\mathtt{rec}(t.\tau)/t]\tau}{\Omega \vdash \mathtt{fold}\{t.\tau\}(e) : \mathtt{rec}(t.\tau)} \tag{C.14g}$$

$$\frac{\Omega \vdash e : \mathtt{rec}(t.\tau)}{\Omega \vdash \mathtt{unfold}(e) : [\mathtt{rec}(t.\tau)/t]\tau} \tag{C.14h}$$

$$\frac{\{\Omega \vdash e_i : \tau_i\}_{i \in L}}{\Omega \vdash \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) : \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{C.14i}$$

$$\frac{\Omega \vdash e : \mathtt{prod}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)}{\Omega \vdash \mathtt{prj}[\ell](e) : \tau} \tag{C.14j}$$

$$\frac{\{\Omega \vdash \tau_i \ \mathsf{type}\}_{i \in L} \qquad \Omega \vdash \tau \ \mathsf{type} \qquad \Omega \vdash e : \tau}{\Omega \vdash \mathtt{inj}[L, \ell; \ell]\{\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau\}(e) : \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)} \tag{C.14k}$$

$$\frac{\Omega \vdash e : \tau \qquad \Omega \vdash \tau' \ \mathsf{type} \qquad \{\Omega \vdash r_i : \tau \Rightarrow \tau'\}_{1 \le i \le n}}{\Omega \vdash \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \le i \le n}) : \tau'} \tag{C.14l}$$

$$\frac{\Omega \vdash M \ \mathsf{mval} \qquad \Omega \vdash M : \mathtt{sig}\{\kappa\}(u.\tau)}{\Omega \vdash \mathtt{val}(M) : [\mathtt{con}(M)/u]\tau} \tag{C.14m}$$

$\boxed{\Omega \vdash r : \tau \Rightarrow \tau'}$ r takes values of type τ to values of type τ'

$$\frac{\Omega \vdash p : \tau \dashv\Vdash \Omega' \qquad \Omega \cup \Omega' \vdash e : \tau'}{\Omega \vdash \mathtt{rule}(p.e) : \tau \Rightarrow \tau'} \tag{C.15}$$

$\boxed{\Omega \vdash p : \tau \dashv\Vdash \Omega'}$ p matches values of type τ generating hypotheses Ω'

$$\frac{\Omega \vdash p : \tau \dashv\Vdash \Omega' \qquad \Omega \vdash \tau <: \tau'}{\Omega \vdash p : \tau' \dashv\Vdash \Omega'} \tag{C.16a}$$

$$\frac{}{\Omega \vdash x : \tau \dashv\Vdash x : \tau} \tag{C.16b}$$

$$\frac{}{\Omega \vdash \mathtt{wildp} : \tau \dashv\Vdash \emptyset} \tag{C.16c}$$

$$\frac{\Omega \vdash p : [\mathtt{rec}(t.\tau)/t]\tau \dashv\Vdash \Omega'}{\Omega \vdash \mathtt{foldp}(p) : \mathtt{rec}(t.\tau) \dashv\Vdash \Omega'} \tag{C.16d}$$

$$\frac{\{\Omega \vdash p_i : \tau_i \dashv\Vdash \Omega_i\}_{i \in L}}{\Omega \vdash \mathtt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\Vdash \cup_{i \in L} \Omega_i} \tag{C.16e}$$

$$\frac{\Omega \vdash p : \tau \dashv\Vdash \Omega'}{\Omega \vdash \mathtt{injp}[\ell](p) : \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\Vdash \Omega'} \tag{C.16f}$$

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. □

Lemma C.2 (Substitution)**.**

1. *If $\Omega, X : \sigma \cup \Omega' \vdash J$ and $\Omega \vdash M : \sigma$ and $\Omega \vdash M$ mval then $\Omega \cup [M/X]\Omega' \vdash [M/X]J$.*
2. *If $\Omega, u :: \kappa \cup \Omega' \vdash J$ and $\Omega \vdash c :: \kappa$ then $\Omega \cup [c/u]\Omega' \vdash [c/u]J$.*
3. *If $\Omega, x : \tau \cup \Omega' \vdash J$ and $\Omega \vdash e : \tau$ then $\Omega \cup [e/x]\Omega' \vdash [e/x]J$.*

Proof Sketch. By straightforward mutual rule induction. □

Lemma C.3 (Decomposition)**.**

1. *If $\Omega \cup [M/X]\Omega' \vdash [M/X]J$ and $\Omega \vdash M : \sigma$ then $\Omega, X : \sigma \cup \Omega' \vdash J$.*
2. *If $\Omega \cup [c/u]\Omega' \vdash [c/u]J$ and $\Omega \vdash c :: \kappa$ then $\Omega \cup \Omega' \vdash J$.*
3. *If $\Omega \cup [e/x]\Omega' \vdash [e/x]J$ and $\Omega \vdash e : \tau$ then $\Omega \cup \Omega' \vdash J$.*

Proof Sketch. By straightforward mutual rule induction. □

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 val	M is a module value
M 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 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.4 (Iterated Transition)**.** *Iterated transition, $e \mapsto^* e'$, is the reflexive, transitive closure of the transition judgement, $e \mapsto e'$.*

Definition C.5 (Evaluation)**.** *$e \Downarrow e'$ iff $e \mapsto^* e'$ and e'* val*.*

As in miniVerse$_\mathbf{S}$, 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.6 (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.7 (Progress)**.**

1. *If $\vdash M : \sigma$ then either M* val *or M* matchfail *or there exists an M' such that $M \mapsto M'$.*
2. *If $\vdash e : \tau$ then either e* val *or e* matchfail *or there exists an e' such that $e \mapsto e'$.*

C.2 Unexpanded Language (UL)

C.3 Proto-Expansion Validation

C.4 Metatheory