Modularly Programmable Syntax

Cyrus Omar

June 2, 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, type systems, module systems, pattern matching, elaboration, bidirectional typing, macro systems, hygiene, parsing, domain-specific languages

TODO: dedication

Abstract

When designing a human-facing textual syntax for a programming language, it is common practice to define derived forms for standard library constructs. For example, the textual syntax of Standard ML (SML) defines derived forms for constructing and pattern matching on lists.

Third-party library providers are, justifiably, envious of this arrangement – although it is straightforward to define other useful derived forms (e.g. for finite maps, HTML elements or regular expressions) in separately developed "domain-specific" dialects of the textual syntax, combining these dialects, when possible at all, is not guaranteed to conserve syntactic determinism (i.e. syntactic conflicts can arise.) Moreover, it can become difficult for programmers to reason in a disciplined manner about types and binding when examining the text of a program that uses unfamiliar derived forms.

In this work, we introduce and formally define *typed syntax macros (TSMs)*, which reduce the need for *ad hoc* derived forms and domain-specific syntax dialects by giving library providers the ability to programmatically control the parsing and expansion, at compile-time, of expressions and patterns of a generalized literal form. Library clients can use any combination of TSMs in a program without needing to consider the possibility of syntactic conflict between them 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:

- a *type discipline*, meaning that the programmer can reason about types without examining the full literal expansion; and
- a *hygienic binding discipline*, meaning that
  1. literal expansions cannot (accidentally or intentionally) shadow bindings that appear at the application site; and
  2. literal expansions cannot refer to application site bindings directly. Instead, all interactions with application site bindings go through either *spliced subterms* and *parameters*.

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

Acknowledgments

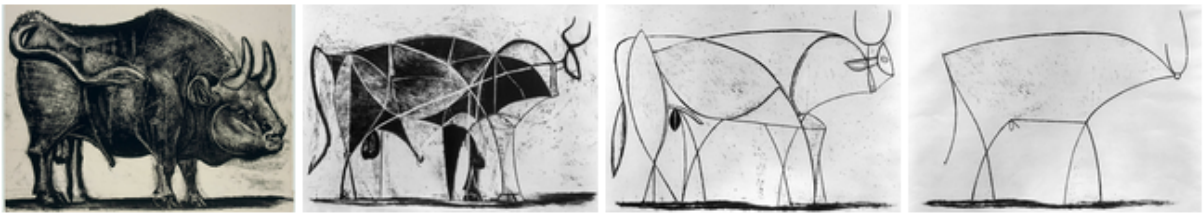
TODO: acknowledgments

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 general, formal structures.

For example, programming language designers often need various sorts of ordered tree structures equipped with metaoperations[1] 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, 38]. 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.

As a simple example, 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 we need to define its semantics and prove it correct. Harper gives a complete account of ABTs, and many other examples of their use, in his textbook [38].

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

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
		$\texttt{plus}(e;e)$	$e + e$	$e\ \texttt{+}\ e$	addition
		$\texttt{mult}(e;e)$	$e \times e$	$e\ \texttt{*}\ e$	multiplication
		$\texttt{div}(e;e)$	$\frac{e}{e}$	$e\ \texttt{/}\ e$	division
		$\texttt{pow}(e;e)$	e^e	$e\texttt{\^}e$	exponentiation

Figure 1.1: Syntax of **Calc**. Metavariable n ranges over mathematical numbers (defined in some suitable manner) and `n` abbreviates the numeral forms (one for each 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 form for parenthesized expressions; 3) defining the precedence and associativity of each infix operator; and 4) defining whitespace conventions.

1.1.1 Syntax Matters

The only problem with this approach is that drawing **Calc** expressions in operational form is rather unwieldy:

$$\texttt{div(num[1];pow(num[2];div(num[1];num[2])))} \tag{1.1a}$$

This is an example of a common problem: instantiating a general-purpose abstraction, here for defining ABTs, can be structurally economical but *syntactically costly* (or *cognitively costly* in some other sense, as we will discuss in Section 2.2.) Mathematics is ultimately a human activity, so these costs are worthy of consideration.

Within a document intended only for human consumption, the usual convention is to informally outline less costly alternative forms. For example, the middle columns of the syntax chart in Figure 1.1 suggest that we can draw any ABT of sort CalcExp, like the example above, instead in an alternative *stylized form*:

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

or in an alternative *textual form*:

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

We might 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 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, 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.) This judgement is defined by a single inference rule, which establishes that every number expression is a value:[2]

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

Although this rule is drawn using the operational form for number expressions, we can apply it to derive that `2` val, because `2` and `num`[2] are structurally identical.

1.1.2 Derived Forms in General-Purpose Languages

Calc is semantically limited - we can express only simple computations of a single type - 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 ABTs, can be adequately expressed within contemporary "general-purpose" languages. Consequently, the problems of syntactic and cognitive cost just discussed also arise "one level down", i.e. when writing programs. For example, we want natural syntax not only for **Calc** expressions expressed as ABTs, but also for encodings of **Calc** expressions expressed using an ABT library for a general-purpose language (e.g. [4] and [5] for Standard ML.)

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 especially difficult, and there are many *syntax definition systems* that help designers 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

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

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 [36, 55]. In SML, the derived expression form `[x, y, z]` desugars to an expression equivalent to:[3]

```
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, e.g.:

- OCaml defines derived forms for mutable arrays and 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.
- F#, Scala and various other languages define derived forms for encodings of the language's own terms (these are referred to as *quasiquotation* forms [66].)
- 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. Indeed, the OCaml community has moved to de-emphasize the standard library in favor of competing bundles of third-party libraries (e.g. Batteries Included [1] is an open source effort, and Core [2] 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 speak more generally about the problems that these existing mechanisms present, to motivate our novel contributions in this area.

[3]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.

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

Figure 1.2: Derived XHTML forms in Ur/Web

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 [18]. 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.

Syntax definition systems like Camlp4 [48], Copper [73] and SugarJ/Sugar* [23, 24], 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.

Some 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 [72]. However, this "dialect-oriented approach" is difficult to reconcile with the best practices of "programming in the large" [20], 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. We summarize the problems that tend to arise 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 typically make use of a variety of libraries and cannot be expected to fall cleanly into preconceived "problem domains".

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 mature syntax definition systems, each differing subtly from the others. If the dialect designers have not chosen the same syntax definition system, the notion of "dialect combination" remains strictly informal.

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 [29]), 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 >}.[4] Though each dialect defines a deterministic grammar, i.e. $\det(\mathcal{D}_1)$ and $\det(\mathcal{D}_2)$, 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 [73], that "nearly" guarantees that determinism is conserved when syntax dialects (of a certain restricted class) are combined [65], 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 Standard ML's textual syntax with a large number of other syntax dialects:

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:

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

```
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.

1. Which constituent dialect defined the derived form that appears on Line 4?
2. Is the character `x` inside this derived form parsed as a "spliced" expression, `x`, or parsed in some other way peculiar to this derived 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?

The problem is fundamentally that 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 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 purpose of using derived forms – lowering cognitive cost.

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 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. As Reynolds famously stated [63]:

   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 Sec. 2.4.6.

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 directly.)

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. 2.4.6, address the problems of 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 [54]. That said, some progress has also been made on this front with the design of *typed macro systems*, like Scala's macro system [16], 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 serve other purposes, leading to confusion on the part of client programmers [61].

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*. For example, in Figure 1.4 we apply a TSM named `$html` to a generalized literal form delimited by backticks. TSM names must be prefixed by `$` and TSMs are applied in post-fix position to clearly distinguish TSM application from function application.

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

Figure 1.4: An example of a TSM being applied to a generalized literal form. The literal body, in blue, 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 blue in Figure 1.4) are otherwise syntactically unconstrained. We use an Ur/Web-inspired HTML syntax (see Figure 1.2) in the literal body in Figure 1.4.

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. As such, the semantics can take the type and binding structure of the surrounding program into account when validating the expansion that the TSM programmatically generates to ensure that clients can answer critical questions related to types and binding, like those enumerated in Section 1.2.1, without having complete knowledge of the implementation of the TSM or the generated expansion.

The primary technical challenge has to do with the fact that the applied TSM needs to be able to parse out subterms from the literal body for inclusion in the expansion. We refer to these as *spliced subterms*. For example, Figure 1.5 reveals the positions of the spliced subterms for the example in Figure 1.4.

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

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

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

We have designed our system so that a figure like this, which presents a *segmentation* of each literal body into spliced subterms (in black) and characters parsed in some other way by the applied TSM (in blue), can always be automatically generated no matter how each applied TSM is implemented.

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. 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

The remainder of this document is organized as follows.

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 very 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 and 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 access "helper" constructs in a controlled and reasonable manner; and 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 can decrease the syntactic cost of this explicit parameter passing style.

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 that we develop in this chapter 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 `string list` type, we can express the example from Figure 1.5 instead as shown in Figure 1.6, assuming `join` has type `string -> string list -> string`. This method is syntactically competitive with library-specific syntax dialects (e.g. compare Figure 1.6 to Figure 1.2), while maintaining the reasoning principles characteristic of our approach.

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

Figure 1.6: The example from Figure 1.5 drawn to take advantage of TSM implicits.

The examples that we will give later are written in a full-scale functional language called VerseML.[5] 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 [22], 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.

[5]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.

1.3.2 Thesis Statement

In summary, this work defends the following statement:

> A programming language with non-trivial type and binding structure 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 the example of languages that support operator overloading or *ad hoc* polymorphism using type classes [21, 35], 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

2.1 Preliminaries

This work is rooted in the tradition of full-scale functional languages with non-trivial type structure 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* [36] (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* [36] as needed. It is important to 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) [12].)

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), renaming, alpha-equivalence, substitution, structural induction and rule induction are defined as described in Harper's *Practical Foundations for Programming Languages, Second Edition* (*PFPL*) [38], except as otherwise stated. Familiarity with other formal accounts of type systems, e.g. Pierce's *Types and Programming Languages* (*TAPL*) [62], should also suffice. This document is organized so as to be readable even if the sections defining formal systems are skipped entirely (although much precision will, of course, be lost).

2.2 Cognitive Cost

The notion of *cognitive cost* is central to our motivations (though it does not enter into any of the technical material directly.) Ultimately, this broad notion must 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 have little hope of developing a comprehensive quantitative framework capable of settling questions related to cognitive cost.[1] Instead, we must turn to frameworks that are merely useful, but not comprehensive [14].

Notions of cognitive cost can perhaps be understood by informal analogy to notions of *dynamic cost*, which distinguish semantically equivalent expressions based on their consumption of various resources, e.g. time or memory, as they are evaluated. Notions of cognitive cost analagously capture the consumption of human attentional resources as they are being drawn and examined by a human. Human attention resources are, of course, more difficult to quantify.

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 usually easier to comprehend and produce. For example, the drawing `[1, 2, 3, 4, 5]` 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 shown in Sec. 1.2.1, 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 introduced by Card, Moran and Newell [17].

One can also analyze cognitive cost using disciplined qualitative methods. For example, Green's *Cognitive Dimensions of Notations* [32, 33] and Pane and Myers' *Usability Issues* [61] (both of which synthesized much of the earlier work in the area) are highly cited heuristic frameworks. For example, Green's cognitive dimensions framework gives us a common vocabulary for comparing the derived list forms described in Chapter 1 to the primitive list forms. In particular, the derived list forms *map more closely* to other notations used for sequences of elements (e.g. in typeset mathematics, or on a

[1] The fact that cognitive cost cannot be comprehensively characterized seems itself to be 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.

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 drawn in primitive form, one needs also to add a closing parenthesis to the end of the term.

Finally, one might consider cognitive cost comparatively using 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 studies, in [68].)

There is much that remains to be understood about cognitive cost, particularly when the subject is an experienced programmer using a language in the functional tradition. 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 drastically 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, so we will err on the side of reasonability.

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

In Standard ML, list types arise out of the following parameterized recursive datatype declaration:

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

This declaration is semantically dense, in that it generates 1) a new type function `list` taking a single type parameter, `'a`; 2) the list value constructors `Nil : 'a list` and `Cons : 'a * 'a list -> 'a list`; and 3) the corresponding list pattern constructors `Nil` and `Cons`.

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 orthogonally. 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.

In VerseML lists are defined as follows:

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

Here, `list` is a type function binding its type parameter to the type variable `'a`. Parameters are applied in prefix position, as in SML. For example, the type of integer lists is `int list`, which is equivalent, by substitution, to the following *recursive type*:

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

The *unfolding* of a recursive type is determined by substituting the recursive type itself for the self reference, here `self`, in the type body. For example, the unfolding of `int list` is equivalent to the following:

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

This *labeled sum type* specifies two *variants*. One, labeled `Nil`, takes values of unit type (we could have written `Nil of unit`.) The other, labeled `Cons`, takes values of the product type `int * int list`.

The values of a recursive type `T` are `(fold e) : T`, where `e` is a value of the unfolding of `T`, and the values of a labeled sum type `T` are `(inj[lbl] e) : T`, where `lbl` is a label specified by one of the variants that `T` specifies, and `e` is a value of the corresponding type. In both cases, the type ascription can be omitted from the program text when it can be inferred. The values of unlabeled product types like `int * int list` are tuples and the only value of unit type is the trivial value `()`, as in Standard ML. For example we can define the empty integer list and bind it to `nil_int` as follows:

```
val nil_int : int list = fold(inj[Nil] ())
```

and we can introduce a list containing a single integer, `0`, and bind it to `one_int` as follows:

```
val one_int : int list = fold(inj[Cons] (0, nil_int))
```

One way to lower syntactic cost is to define the following polymorphic values, called the *list value constructors*, which abstract away the fold and injection operations:

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

In fact, VerseML generates constructors like these automatically.[2] Using these constructors, we can equivalently express the bindings of `nil_int` and `one_int` as follows:

```
val nil_int : int list = Nil
val one_int = Cons (0, Nil)
```

In SML, automatically generated constructors are the only means by which a value of a datatype can be introduced. Folding and injection operators are not exposed directly to programmers. As such, it is not possible to construct a value of a type like `int list` in a context-independent manner, i.e. in contexts where `Nil` and `Cons` have been shadowed or are not bound. This will be relevant in the next section and in Chapter 3 and Chapter

[2]The mechanism for automatically generating value constructors from type bindings of certain forms must be built primitively into VerseML. A more general mechanism that allowed a library provider to generate such bindings implicitly would make it difficult to reason about shadowing.

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

Figure 2.1: Definition of the `LIST` signature.

4. In Chapter 5, we will introduce the machinery that would be necessary to take the SML-style approach and suppress mention of **`fold`** and **`inj`** operators entirely.

Values of recursive type, labeled sum type and product type are deconstructed by pattern matching.[3] For example, we can write the polymorphic map function, which constructs a list by applying a given function over a given list, as follows:

```
fun map (f : 'a -> 'b) (xs : 'a list) : 'b list =>
  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:[4]

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

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

[3]Readers who are not familiar with structural pattern matching may wish to consult the introduction to Chapter 4 for a somewhat more detailed description.

[4]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* [70]. Regexes are common in domains 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. 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 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 implement regexes, each with different performance trade-offs, so we would like to provide a choice of implementations 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 [52].) In particular, we define the signature abbreviated `RX` in Figure 2.4.

```
type 'a u = UEmpty + UStr of string + USeq of 'a * 'a +
            UOr of 'a * 'a + UStar of 'a

signature RX =
sig
  type t (* abstract *)

  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 -> t u
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) [49].) Consequently, representation invariants need only be established locally within each module.

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*[5] 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:

```
fun is_dna_rx'(r : R.t) : boolean =>
  match R.unfold_norm r with
  | UStr "A" => True
```

[5]This sense of the word "unfolding" is conceptually related to, but technically distinct, from the sense in which it is used for recursive types discussed above.

```
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 2.5: The definition of `RXUtil`.

```
  | 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
```

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.

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.[6]

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 2.5, 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)
```

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

2.4 Existing Approaches

The constructs defined in the previous section adequately encode the semantics of lists and regular expressions. Our next task is to define auxiliary constructs that help to decrease the syntactic cost of expressions and patterns involving these semantically central constructs, particularly those that arise the most frequently (i.e. *idiomatic* expressions and patterns) and that have the highest syntactic cost. There are many approaches that a library provider might consider. These differ with regard to how much control they give the library provider over syntactic form, and the broader cognitive burdens that they impose on client programmers, particularly those programming in the large.

2.4.1 Standard Abstraction Mechanisms

The simplest approach 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 2.5. 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 way to capture idioms at the level of patterns.

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 definitions like those above (or else SML would not have needed to primitively define derived list forms.) Similarly, consider the textual syntax for regexes defined in the POSIX standard [8]. 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 previously, and programmers familiar with the POSIX syntax would likely agree that these drawings have lower cognitive cost as well. Data suggests that most professional programmers are familiar with these forms [58].

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:[7]

   ```
   error: illegal escape character
   ```

   In a small lab study, we observed that even experienced programmers made this mistake and could not quickly diagnose the problem and determine a workaround if they had not used a regex library recently [58].

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

   ```
   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.

   ```
   {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 2.6 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.

   We will describe derived forms that do capture the idioms of compositional regex construction in Sec. 2.4.5 (in particular, we will compare Figures 2.6 and 2.8.)

[7] 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`.

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

Figure 2.6: Compositional construction of a regex.

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 seemingly "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 a regex that matches *other* strings that it behaves incorrectly.

   In applications that query sensitive data, mistakes like this lead to *injection attacks*, which are among the most common and catastrophic security threats today [9].

   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 [15].

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" [67].

   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. This subtly changes the performance profile of the program, and there is a cognitive penalty associated with moving the de-

scription 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 [67].

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 regular expressions 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 the SML/NJ compiler [6], 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"]
```

This gives library providers substantial control over form, and, unlike dynamic string parsing, allows library providers to capture idioms involving subexpressions. For example, the regex library provider can add a function `qparse : R.t frag list -> R.t` to `RXUtil` that parses the given fragment list, 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 low syntactic cost.

The list library provider could also add a function `qparse : 'a frag list -> 'a list` to the `List` module that constructs a list from a quoted list:

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

As with dynamic string parsing, parsing occurs dynamically. We cannot use the trick of lifting all calls to `qparse` to the top level of our program, because the arguments are no longer string literals. At best, we can lift these calls out as far as the binding structure allows, i.e. into the earliest possible "phase of evaluation". Parse errors in quoted parts of the quotation literal are only detected 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 (x => List.qparse `[^x, ^(2 * x)]`) input
```

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.

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 [48]), clients would need to apply marking constructors to each element of the list:

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

Marking constructors increase syntactic cost rather substantially in these examples.

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.)

VerseML does not build in quotation literals.[8]

2.4.3 Fixity Directives

To gain more precise control over form, a library provider, or another interested party, can define a "library-specific" syntax dialect using a *syntax definition system*.

The simplest syntax definition systems allow programmers to designate identifiers as 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`. If `op::` is bound to the list constructor `Cons`, 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. Unfortunately, the definition of `list` that we gave in Sec. 2.3.1 introduced a pattern constructor identified as `Cons`, not `op::`, so this infix form does not desugar to a valid list pattern. There is no mechanism for defining pattern constructor abbreviations in SML. The SML Basis library uses the label `op::` rather than `Cons` directly in its definition of the list datatype for this reason.

Figure 2.7 shows three fixity declarations related to our regex library together with a functor `RXOps` that binds the corresponding identifiers to the appropriate functions. Assuming that a library packaging system has brought the fixity declarations and the definition of `RXOps` from Figure 2.7 into scope, we can instantiate `RXOps` and then **open** this instantiated module to bring the necessary bindings into scope as follows:

[8] 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
  module RU = RXUtil(R)
  val op:: = R.Seq
  val op<*> = RU.repeat
  val op<|> = R.Or
end
```

Figure 2.7: Fixity declarations and related bindings for `RX`.

```
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 the two main 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 meaning.)

2.4.4 Mixfix Syntax Definitions

Fixity directives do not give programmers direct control over desugaring – in SML, 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 rule.

The simplest of these systems, e.g. Griffin's early work on notational definitions [34] and the syntax definition system integrated into the Agda programming language [19], support only forms that contain a fixed number of sub-trees, e.g. **`if _ then _ else _`**. SML-style derived list forms cannot be defined using these systems, because lists can contain any number of sub-expressions.

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 [53] 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 [53]:

> *Mixing different symbolic notations in a same text may cause serious parsing ambiguity.*

To help library clients manage the problem of conflict, most of these systems include various generalized precedence mechanisms. For example, Agda supports a system of directed acyclic precedence graphs [19] (this is related to earlier work by Aasa where a complete precedence graph was necessary [10].) 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.

2.4.5 More General Syntax Definition Systems

More general syntax definition systems give users more control over form than the infix and mixfix systems just described. In this section, we will begin by describing some notable systems, then give two examples that demonstrate their expressive power. This is followed by a discussion of the difficulties that programmers can expect to encounter if they use these systems for programming in the large (as a follow-up to what was discussed in Section 1.2.1.)

Most of the systems that we will describe in this section operate as language-external *preprocessors*, transforming source text into programs or program fragments. Some compilers, e.g. the OCaml compiler [48], 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 [29]. Only a few systems are integrated more directly into the language definition – we will point these out when we introduce them.

Grammar-Based Syntax Definition Systems

Many syntax definition systems are oriented around *formal grammars* [42]. 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 [45].

Context-free grammars (CFGs) were first used to define the textual syntax of a major programming language – Algol 60 – by Backus [56]. 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 scanner or lexer definition in most systems) and logic for computing an output value (e.g. a tree) based on the determined form of the input text.

Perhaps the most established CFG-based syntax definition systems within the ML ecosystem are ML-Lex and ML-Yacc, which are distributed with SML/NJ [69], 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) [48]. 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)* [40] is a syntactic formalism for describing CFGs. SDF is used by a number of syntax definition systems, e.g. the Spoofax "language workbench" [46]. These systems commonly use Stratego, a rule-based rewriting language, as the language that output logic is written in [71]. 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 [24]. SugarHaskell is a similar system based on Haskell [26] and Sugar* simplifies the task of defining similar extensions of other languages [23]. SoundExt and SugarFOmega add the requirement that new derived forms must come equipped with derived typing rules [50]. 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 [28].) SoundX generalizes this idea to other base languages, and adds the ability to define type-dependent rewritings [51]. We will say more about SoundExt/SugarFOmega and SoundX when we discuss abstract reasoning under syntax dialects shortly.

Copper implements a CFG-based syntax definition system that uses a context-aware scanner [73]. We will say more about Copper when we discuss modular reasoning about

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

Figure 2.8: Derived regex expression forms in $\mathcal{V}_{\texttt{rx}}$

syntactic determinism shortly.

Some other syntax definition systems are instead oriented around *parsing expression grammars* (PEGs) [30]. PEGs are similar to CFGs, distinguished mainly in that they are deterministic by construction (by allowing only for explicitly prioritized choice between alternative parses.)

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*) [43]. 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` is declared as follows:

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

`alt` 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 [44]. Okasaki has described an alternative design that uses continuations to control cost [57].

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

Using the more general syntax definition systems just described, we can define a dialect of VerseML's textual syntax called $\mathcal{V}_{\texttt{rx}}$ that builds in derived regex forms following the POSIX standard.[9]

In particular, $\mathcal{V}_{\texttt{rx}}$ extends the syntax of expressions with *derived regex literals*, which are delimited by forward slashes, e.g.:

[9]Technically, $\mathcal{V}_{\texttt{rx}}$ is a dialect of the textual syntax of VerseML's inner core language; 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 2.9: Derived regex pattern forms in $\mathcal{V}_{\texttt{rx}}$

```
/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.[10]

$\mathcal{V}_{\texttt{rx}}$ 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 2.8 is equivalent to the definition of `lookup_rx` that was given in Figure 2.6, 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 2.9 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 2.9 appear inside *spliced sub-patterns*.

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

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

In Sec. 2.3.2, we gave 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\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 is context-independent because the constructors are explicitly qualified (i.e. `Seq` and `Str` are *field labels* here, not variables.) The only variables that appear in the desugaring are `R`, `name` and `ssn`. All of these also appeared at the use site.

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 `R.t u`. $\mathcal{V}_{\text{RX}}$ defines derived forms for patterns matching values of types in the type family `'a u`. These appear in the definition of `is_dna_rx'` given in Figure 2.10.

Combining $\mathcal{V}_{\text{rx}}$ and $\mathcal{V}_{\text{RX}}$

Notice that the derived regex pattern forms that appear in Figure 2.10 are identical to those that appear in Figure 2.9. Their desugarings are, however, different. In particular, the patterns in Figure 2.10 match values of type `R.t u`, whereas the patterns in Figure 2.9 match values of type `rx`.

[10] 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.

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 [30].)

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 [65]. By relying on a context-aware scanner (a feature of Copper [73]) 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 given in [65] 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 2.11 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 2.10. 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.

TODO: mention this? `http://www.ccs.neu.edu/home/ejs/papers/tfp12-island.pdf`

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

```
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 2.11: Using URI-based grammar names together with marking tokens to avoid syntactic conflicts.

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 [65] 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 [46] and Sugarclipse for SugarJ [25], 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 introduce locally-scoped bindings and, simultaneously, exported module components that are similarly invisible in the text. This can cause both local shadowing as well as 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. If the types of the two bindings are incompatible, the problem will be caught statically. Otherwise, it will cause unanticipated dynamic behavior. It is easy to miss this problem in testing.

   In some syntax dialects, shadowing is by design. For example, in (Sugar)Haskell, **`do`** notation for monadic values introduces a new binding construct [26]. 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 not guarantee that the dialect provider maintained this freshness discipline [27].

   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 [53], 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 (and more generally, a rewriting) to avoid shadowing [27, 64].[11]

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

[11] 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.

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 [51] and SoundX [64] 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 derivation requires significantly more expertise than programmers usually need.[12] For languages like ML, the judgement forms are substantially more complex.

As discussed in Sec. 1.2.1, languages with a rich type and binding structure are designed to minimize or eliminate the cognitive cost of answering analagous questions. These reasoning principles are central to the claim that such languages are suitable for "programming in the large" [20].

Due to the problems of syntactic conflict and the reasoning difficulties enumerated above, the textual syntax of VerseML cannot be modified or extended from within.

[12] 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.)

2.4.6 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*.

Language-External Term Rewriting Systems

Language-external rewriting systems operate as *preprocessors*, transforming well-formed program fragments to produce other well-formed program fragments.

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`, following the approach discussed in the previous section. 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 [**?**]. Extension points serve as markers for the benefit of a preprocessor. 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.

These systems present conceptual problems that are directly analagous to those that dialect-oriented systems present:

1. **Conflict:** Different preprocessors may recognize the same markers.
2. **Search:** It is not always clear which preprocessor handles each marked form.
3. **Segmentation:** It is not always clear where spliced sub-terms appear inside marked forms (particularly string literals).
4. **Shadowing:** The rewriting of a marked form might place terms under binders that shadow bindings visible in the program text.
5. **Context Dependence:** The rewriting of a marked form might assume that certain identifiers are bonud at the use site, making it difficult to reason about refactoring.
6. **Typing:** It is not always clear what type the rewriting of a marked form will have.

Macro Systems

Macro systems allow programmers to designate functions that operate over term encodings as macros, and then apply these macros directly to terms as rewritings.

The LISP macro system [39] is perhaps the most prominent example of such a system. Early variants of this system suffered from the problem of hygiene described earlier TODO: elaborate, but later variants, notably in the Scheme dialect of LISP, brought support for enforcing hygiene [47].

In languages with a richer static type discipline, variants of macros that restrict rewriting to a particular type and perform the rewriting statically have also been studied [31, 41] and integrated into languages, e.g. MacroML [31] and Scala [16]. TODO: actually MacroML isn't a term rewriting system; just a staging system. write about this.

The most immediate problem with using these for our example is that we are not aware of any such statically-typed macro system that integrates cleanly with an ML-style module system. In other words, macros cannot be parameterized by modules. However, let us imagine such a macro system. We could use it to repurpose string syntax as follows:

```
let val ssn = rx R {rx|\d\d\d-\d\d-\d\d\d\d|rx}
```

The definition of the macro `rx` might look like this:

```
1 macro rx[Q : RX](e) at Q.t {
2   static fun f(e : Exp) : Exp => case(e) {
3       StrLit(s) => (* regex parser here *)
4     | BinOp(Caret, e1, e2) => `Q.Seq(Q.Str(%e1), %(f e2))`
5     | BinOp(Plus, e1, e2) => `Q.Seq(%(f e1), %(f e2))`
6     | _ => raise Error
7   }
8 }
```

Here, `rx` is a macro parameterized by a module matching `rx` (we identify it as `Q` to emphasize that there is nothing special about the identifier `R`) and taking a single argument, identified as `e`. The macro specifies a type annotation, `at Q.t`, which imposes the constraint that the expansion the macro statically generates must be of type `Q.t` for the provided parameter `Q`. This expansion is generated by a *static function* that examines the syntax tree of the provided argument (syntax trees are of a type `Exp` defined in the standard library; cf. SML/NJ's visible compiler [7]). If it is a string literal, as in the example above, it statically parses the literal body to generate an expansion (the details of the parser, elided on line 3, would be entirely standard). By parsing the string statically, we avoid the problems of dynamic string parsing for statically-known patterns.

For patterns that are constructed compositionally, we need to get more creative. For example, we might repurpose the infix operators that are normally used for other purposes to support string and pattern splicing, e.g. as follows:

```
fun example_using_macro(name : string) =>
  rx R (name ^ ": " + ssn)
```

The binary operator `^` is repurposed to indicate a spliced string and `+` is repurposed to indicate a spliced pattern. The logic for handling these forms can be seen above on lines 4 and 5, respectively. We assume that there is derived syntax available at the type `Exp`, i.e. *quasiquotation* syntax as in Lisp [13] and Scala [66], here delimited by backticks and using the prefix `%` to indicate a spliced value (i.e. unquote).

Having to creatively repurpose existing forms in this way limits the effect a library provider can have on cognitive cost (particularly when it would be desirable to express conventions that are quite different from the conventions adopted by the language). It also can create confusion for readers expecting parenthesized expressions to behave in a consistent manner. However, this approach is preferable to direct syntax extension because it avoids many of the problems discussed above: there cannot be syntactic conflicts (because the syntax is not extended at all), we can define macro abbreviations because macros are integrated into the language, there is a hygiene discipline that guarantees that the expansion will not capture variables inadvertently, and by using a typed macro system, programmers need not examine the expansion to know what type the expansion produced by a macro must have.

Part I

Simple TSMs

Chapter 3

Unparameterized Expression TSMs (ueTSMs)

We now introduce a new primitive – the **typed syntax macro** (TSM). TSMs, like term-rewriting macros (Sec. 2.4.6), generate expansions. Unlike term-rewriting macros, TSMs are applied to unparsed *generalized literal forms*, which gives them substantially more syntactic flexibility. This chapter considers perhaps the simplest manifestation of TSMs: **unparameterized expression TSMs** (ueTSMs), which generate expressions of a single specified type. We will consider unparameterized pattern TSMs (upTSMs) in Chapter 4 and parameterized TSMs (pTSMs) in Chapter 5.

3.1 Expression TSMs By Example

We begin in this section with a "tutorial-style" introduction to ueTSMs in VerseML. In particular, we discuss a ueTSM for constructing values of the recursive labeled sum type `Rx` that was defined in Figure 2.2. We then formally specify ueTSMs with a reduced calculus, $\text{miniVerse}_{\text{UE}}$, in Sec. 3.2.

3.1.1 Usage

In the following textual VerseML expression, we apply a TSM named `$rx` to the *generalized literal form* `/A|T|G|C/`: TODO: make this post-fix

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

Generalized literal forms are left unparsed when concrete expressions are first parsed. It is only during the subsequent *typed expansion* process that the TSM parses the *body* of the provided literal form, i.e. the characters between forward slashes in blue here, to generate a *candidate expansion*. The language then *validates* the candidate expansion according to criteria that we will establish in Sec. 3.1.4. If candidate expansion validation succeeds, the language generates the *final expansion* (or more concisely, simply the *expansion*) of the expression. The program will behave as if the expression above has been replaced by its expansion. The expansion of the expression above, written concretely, is:

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

Figure 3.1: Generalized literal forms available for use in VerseML's concrete syntax. The characters in blue indicate where the literal bodies are located within each form. In this figure, each line describes how the literal body is constrained by the form shown on that line. The Wyvern language specifies additional forms, including whitespace-delimited forms [59] and multipart forms [60], but for simplicity we leave these out of VerseML.

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

A number of literal forms, shown in Figure 3.1, are available in VerseML's concrete syntax. Any literal form can be used with any TSM, e.g. we could have equivalently written the example above as `$rx` `` `A|T|G|C` `` (in fact, this would be convenient if we had wanted to express a regex containing forward slashes but not backticks). TSMs have access only to the literal bodies. Because TSMs do not extend the concrete syntax of the language directly, there cannot be syntactic conflicts between TSMs.

3.1.2 Definition

Let us now take the perspective of the library provider. The definition of the TSM `$rx` shown being applied above has the following form:

```
syntax $rx at Rx {
  static fn(body : Body) : CEExp ParseResult =>
    (* regex literal parser here *)
}
```

This TSM definition first names the TSM. TSM names must begin with the dollar symbol (`$`) to clearly distinguish them from variables (and thereby clearly distinguish TSM application from function application). This is inspired by a similar convention enforced by the Rust macro system [3].

The TSM definition then specifies a *type annotation*, **`at`** `Rx`, and a *parse function* within curly braces. The parse function is a *static function* responsible for parsing the literal body when the TSM is applied to generate an encoding of the candidate expansion, or an indication of an error if one cannot be generated (e.g. when the body is ill-formed according to the syntactic specification that the TSM implements). Static functions are functions that are applied during the typed expansion process. For this reason, they do not have access to surrounding variable bindings (because those variables stand in for dynamic values). For now, let us simply assume that static functions are closed (we discuss introducing a distinct class of static bindings so that static values can be shared between TSM definitions in Sec. 6.2).

```
type IndexRange = {startIndex : nat, endIndex : nat}

type 'a ParseResult = Success of 'a
                    | ParseError of {
                        msg : string, loc : IndexRange
                      }
```

Figure 3.2: Definitions of `IndexRange` and `ParseResult` in the VerseML prelude.

```
type CETyp = TyVar of var_t
           | Arrow of CETyp * CETyp
           | (* ... *)
           | Spliced of IndexRange

type CEExp = Var of var_t
           | Fn of var_t * CETyp * CEExp
           | App of CEExp * CEExp
           | (* ... *)
           | Spliced of IndexRange
```

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

The parse function must have type `Body -> CEExp ParseResult`. These types are defined in the VerseML *prelude*, which is a collection of definitions available ambiently. The input type, `Body`, gives the parse function access to the body of the provided literal form. For our purposes, it suffices to define `Body` as an abbreviation for the `string` type:

```
type Body = string
```

The output type, `CEExp ParseResult`, is a labeled sum type that distinguishes between successful parses and parse errors. The parameterized type `'a ParseResult` is defined in Figure 3.2.

If parsing succeeds, the parse function returns a value of the form `Success(`e_{cand}`)`, where e_{cand} is the *encoding of the candidate expansion*. Encodings of candidate expansions are, for expression TSMs, values of the type `CEExp` defined in Figure 3.3 (in Chapter 5, we will introduce pattern TSMs, which generate patterns rather than expressions; this is why `ParseResult` is defined as a parameterized type). Expressions can mention types, so we also need to define a type `CETyp` in Figure 3.3. We discuss the constructors labeled `Spliced` in Sec. 3.1.3; the remaining constructors (some of which are elided for concision) encode the abstract syntax of VerseML expressions and types. To decrease the syntactic cost of working with the types defined in Figure 3.3, the prelude provides *quasiquotation syntax* at these types, which is itself implemented using TSMs. We will discuss these TSMs in more detail in Sec. 6.1. The definitions in Figure 3.3 are recursive labeled sum types to simplify our exposition, but we could have chosen alternative encodings of terms, e.g. based on abstract binding trees [38], with only minor modification to the semantics.

If the parse function determines that a candidate expansion cannot be generated, i.e. there is a parse error in the literal body, it returns a value labeled by `ParseError`. It must provide an error message and indicate the location of the error within the body of the literal form as a value of type `IndexRange`, also defined in Figure 3.2. This information can be used by VerseML compilers when reporting errors to the programmer.

3.1.3 Splicing

To support splicing syntax, like that described in Sec. 2.3.2, the parse function must be able to parse subexpressions out of the supplied literal body. For example, consider the code snippet in Figure 2.8, expressed instead using the `$rx` TSM:

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

The subexpressions `name` and `ssn` on the second line appear directly in the body of the literal form, so we call them *spliced subexpressions* (and color them black when typesetting them in this document). When the parse function determines that a subsequence of the literal body should be treated as a spliced subexpression (here, by recognizing the characters `@` or `%` followed by a variable or parenthesized expression), it can refer to it within the candidate expansion it generates using the `Spliced` constructor of the `CEExp` type shown in Figure 3.3. The `Spliced` constructor requires a value of type `IndexRange` because spliced subexpressions are referred to indirectly by their position within the literal body. This prevents TSMs from "forging" a spliced subexpression (i.e. claiming that an expression is a spliced subexpression, even though it does not appear in the body of the literal form). Expressions can also contain types, so one can also mark spliced types in an analagous manner using the `Spliced` constructor of the `CETyp` type.

The candidate expansion generated by `$rx` for the body of `example_rx_tsm`, if written in a hypothetical concrete syntax for candidate expansions where references to spliced subexpressions are written `spliced<startIdx, endIndex>`, is:

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

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

3.1.4 Typing

The language *validates* candidate expansions before a final expansion is generated. One aspect of candidate expansion validation is checking the candidate expansion against the type annotation specified by the TSM, e.g. the type `Rx` in the example above. This maintains a *type discipline*: if a programmer sees a TSM being applied when examining a well-typed program, they need only look up the TSM's type annotation to determine the type of the generated expansion. Determining the type does not require examining the expansion directly.

3.1.5 Hygiene

The spliced subexpressions that the candidate expansion refers to (by their position within the literal body, cf. above) must be parsed, typed and expanded during the candidate expansion validation process (otherwise, the language would not be able to check the type of the candidate expansion). To maintain a useful *binding discipline*, i.e. to allow programmers to reason also about variable binding without examining expansions directly, the validation process maintains two additional properties related to spliced subexpressions: **context independent expansion** and **expansion independent splicing**. These are collectively referred to as the *hygiene properties* (because they are conceptually related to the concept of hygiene in term rewriting macro systems, cf. Sec. 2.4.6.)

Context Independent Expansion Programmers expect to be able to choose variable and symbol names freely, i.e. without needing to satisfy "hidden assumptions" made by the TSMs that are applied in scope of a binding. For this reason, context-dependent candidate expansions, i.e. those with free variables or symbols, are deemed invalid (even at application sites where those variables happen to be bound). An example of a TSM that generates context-dependent candidate expansions is shown below:

```
syntax $bad1 at Rx {
  static fn(body : Body) : ParseResultExp => Success (Var 'x')
}
```

The candidate expansion this TSM generates would be well-typed only when there is an assumption `x : Rx` in the application site typing context. This "hidden assumption" makes reasoning about binding and renaming especially difficult, so this candidate expansion is deemed invalid (even when `$bad1` is applied in a context where `x` happens to be bound).

Of course, this prohibition does not extend into the spliced subexpressions referred to in a candidate expansion because spliced subexpressions are authored by the TSM client and appear at the application site, and so can justifiably refer to application site bindings. We saw examples of spliced subexpressions that referred to variables bound at the application site in Sec. 3.1.3. Because candidate expansions refer to spliced subexpressions indirectly, checking this property is straightforward – we only allow access to the application site typing context when typing spliced subexpressions. In the next section, we will formalize this intuition.

In the examples in Sec. 3.1.1 and Sec. 3.1.3, the expansion used constructors associated with the `Rx` type, e.g. `Seq` and `Str`. This might appear to violate our prohibition on context-dependent expansions. This is not the case only because in VerseML, constructor labels are not variables or scoped symbols. Syntactically, they must begin with a capital letter (like Haskell's datatype constructors). Different labeled sum types can use common constructor labels without conflict because the type the term is being checked against – e.g. `Rx`, due to the type ascription on `$rx` – determines which type of value will be constructed. For dialects of ML where datatype definitions do introduce new

variables or scoped symbols, we need parameterized TSMs. We will return to this topic in Chapter 5.

Expansion Independent Splicing Spliced subexpressions, as just described, must be given access to application site bindings. The *expansion independent splicing* property ensures that spliced subexpressions have access to *only* those bindings, i.e. a TSM cannot introduce new bindings into spliced subexpressions. For example, consider the following hypothetical candidate expansion (written concretely as above):

```
fn(x : Rx) => spliced<0, 4>
```

The variable `x` is not available when typing the indicated spliced subexpression, nor can it shadow any bindings of `x` that might appear at the application site.

For TSM providers, the benefit of this property is that they can choose the names of variables used internally within expansions freely, without worrying about whether they might shadow those that a client might have defined at the application site.

TSM clients can, in turn, determine exactly which bindings are available in a spliced subexpression without examining the expansion it appears within. In other words, there can be no "hidden variables".

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 show an alternative formulation of Haskell's syntax for monadic commands that uses VerseML's anonymous function syntax to bind variables in Sec. 8.1.1. We will discuss mechanisms that would allow us to relax this restriction while retaining client control over variable names as future work in Sec. 8.6.6.

3.1.6 Final Expansion

If validation succeeds, the language generates the *final expansion* from the candidate expansion by replacing references to spliced subexpressions with their final expansions. The final expansion of the body of `example_rx_tsm` is:

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

3.1.7 Scoping

A benefit of specifying TSMs as a language primitive, rather than relying on extralinguistic mechanisms to manipulate the concrete 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 { (* ... *) } in
           (* $rx is in scope here *)
         end)
in (* $rx is no longer in scope *) end
```

If the **in** clause is omitted, the scope of the TSM extends to the end of the current block. We will consider the question of how TSM definitions can be exported from compilation units in Sec. 8.3.1.

3.1.8 Comparison to ML+Rx

Let us compare the VerseML TSM `$rx` to ML+Rx, the hypothetical syntactic dialect of ML with support for derived forms for regular expressions described in Sec. 2.3.2.

Both ML+Rx and `$rx` give programmers the ability to use the same standard syntax for constructing regexes, including syntax for splicing in other strings and regexes. In VerseML, however, we incur the additional syntactic cost of explicitly applying the `$rx` TSM each time we wish to use regex syntax. This cost does not grow with the size of the regex, so it would only be significant in programs that involve a large number of small regexes (which do, of course, exist). In Chapter 7 we will consider a design where even this syntactic cost can be eliminated in certain situations.

The benefit of this 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 typing discipline and the hygienic binding discipline described above to reason about programs, including those that contain unfamiliar forms. Put pithily, VerseML helps programmers avoid "conflict and confusion".

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

To make the intuitions developed in the previous section mathematically precise, we will now introduce a reduced language called miniVerse$_{\mathbf{UE}}$ with support for ueTSMs. miniVerse$_{\mathbf{UE}}$ consists of an *inner core* and an *outer surface*.

3.2.1 Syntax of the Inner Core

The *inner core of* miniVerse$_{\mathbf{UE}}$ consists of *types*, τ, and *expanded expressions*, e. The syntax of the inner core is specified by the syntax chart in Figure 3.4. The inner core forms a pure language with support for partial functions, quantification over types, recursive types, labeled product types and labeled sum types. The reader is directed to *PFPL* [38] (or another text on type systems, e.g. *TAPL* [62]) for a detailed introductory account of these (or very similar) constructs. We will tersely define the statics of the inner core, and outline the structural dynamics, in the next two subsections, respectively.

3.2.2 Statics of the Inner Core

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

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

Figure 3.4: Abstract syntax of types and expanded expressions, which form the *inner core of* $\mathsf{miniVerse}_{\mathbf{UE}}$. Metavariables x range over variables, t over type variables, ℓ over labels and L over finite sets of labels. We adopt *PFPL*'s conventions for operational forms, i.e. 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 "phase") using {textual curly braces} and (textual rounded braces) [38]. 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 parameterized by label sets, e.g. $\texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})$, are identified up to mutual reordering of the label set and the corresponding argument sequence. When we use the stylized forms, we assume that the reader can infer suppressed indices and arguments from the surrounding context. Types and expanded expressions are identified up to α-equivalence.

Judgement Form	**Description**
$\Delta \vdash \tau\ \mathsf{type}$	τ is a well-formed type assuming Δ
$\Delta\ \Gamma \vdash e : \tau$	e is assigned type τ assuming Δ and Γ

Type formation contexts, Δ, are finite sets of hypotheses of the form $t\ \mathsf{type}$. 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 $\Delta, t\ \mathsf{type}$, when $t\ \mathsf{type} \notin \Delta$, for Δ extended with the hypothesis $t\ \mathsf{type}$.

The *type formation judgement*, $\Delta \vdash \tau\ \mathsf{type}$, is inductively defined by the following rules:

$$\frac{}{\Delta, t\ \mathsf{type} \vdash t\ \mathsf{type}} \tag{3.1a}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{type} \qquad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \mathtt{parr}(\tau_1; \tau_2)\ \mathsf{type}} \tag{3.1b}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \mathtt{all}(t.\tau)\ \mathsf{type}} \tag{3.1c}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \mathtt{rec}(t.\tau)\ \mathsf{type}} \tag{3.1d}$$

$$\frac{\{\Delta \vdash \tau_i\ \mathsf{type}\}_{i \in L}}{\Delta \vdash \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})\ \mathsf{type}} \tag{3.1e}$$

$$\frac{\{\Delta \vdash \tau_i\ \mathsf{type}\}_{i \in L}}{\Delta \vdash \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L})\ \mathsf{type}} \tag{3.1f}$$

Premises of the form $\{J_i\}_{i \in L}$ mean that for each $i \in L$, the judgement J_i must hold.

Typing contexts, Γ, are finite functions that map each variable $x \in \mathrm{dom}(\Gamma)$, 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 (we do not separately write down the finite set $\mathrm{dom}(\Gamma)$ because it can be determined from the listed hypotheses). 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 $\Delta \vdash \Gamma\ \mathsf{ctx}$ if every type in Γ is well-formed relative to Δ.

Definition 3.1 (Typing Context Formation)**.** $\Delta \vdash \Gamma\ \mathsf{ctx}$ *iff for each hypothesis* $x : \tau \in \Gamma$*, we have* $\Delta \vdash \tau\ \mathsf{type}$*.*

The typing judgement, $\Delta\ \Gamma \vdash e : \tau$, assigns types to expressions. It is inductively defined by the following rules:

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash x : \tau} \tag{3.2a}$$

$$\frac{\Delta \vdash \tau \text{ type} \qquad \Delta\, \Gamma, x : \tau \vdash e : \tau'}{\Delta\, \Gamma \vdash \mathtt{lam\{\tau\}(x.e)} : \mathtt{parr}(\tau; \tau')} \tag{3.2b}$$

$$\frac{\Delta\, \Gamma \vdash e_1 : \mathtt{parr}(\tau; \tau') \qquad \Delta\, \Gamma \vdash e_2 : \tau}{\Delta\, \Gamma \vdash \mathtt{ap}(e_1; e_2) : \tau'} \tag{3.2c}$$

$$\frac{\Delta, t \text{ type}\, \Gamma \vdash e : \tau}{\Delta\, \Gamma \vdash \mathtt{tlam}(t.e) : \mathtt{all}(t.\tau)} \tag{3.2d}$$

$$\frac{\Delta\, \Gamma \vdash e : \mathtt{all}(t.\tau) \qquad \Delta \vdash \tau' \text{ type}}{\Delta\, \Gamma \vdash \mathtt{tap\{\tau'\}}(e) : [\tau'/t]\tau} \tag{3.2e}$$

$$\frac{\Delta, t \text{ type} \vdash \tau \text{ type} \qquad \Delta\, \Gamma \vdash e : [\mathtt{rec}(t.\tau)/t]\tau}{\Delta\, \Gamma \vdash \mathtt{fold\{}t.\tau\mathtt{\}}(e) : \mathtt{rec}(t.\tau)} \tag{3.2f}$$

$$\frac{\Delta\, \Gamma \vdash e : \mathtt{rec}(t.\tau)}{\Delta\, \Gamma \vdash \mathtt{unfold}(e) : [\mathtt{rec}(t.\tau)/t]\tau} \tag{3.2g}$$

$$\frac{\{\Delta\, \Gamma \vdash e_i : \tau_i\}_{i \in L}}{\Delta\, \Gamma \vdash \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) : \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{3.2h}$$

$$\frac{\Delta\, \Gamma \vdash e : \mathtt{prod}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)}{\Delta\, \Gamma \vdash \mathtt{pr}[\ell](e) : \tau} \tag{3.2i}$$

$$\frac{\{\Delta \vdash \tau_i \text{ type}\}_{i \in L} \qquad \Delta \vdash \tau \text{ type} \qquad \Delta\, \Gamma \vdash e : \tau}{\Delta\, \Gamma \vdash \mathtt{in}[L, \ell][\ell]\{\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau\}(e) : \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)} \tag{3.2j}$$

$$\frac{\Delta\, \Gamma \vdash e : \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \qquad \Delta \vdash \tau \text{ type} \qquad \{\Delta\, \Gamma, x_i : \tau_i \vdash e_i : \tau\}_{i \in L}}{\Delta\, \Gamma \vdash \mathtt{case}[L]\{\tau\}(e; \{i \hookrightarrow x_i.e_i\}_{i \in L}) : \tau} \tag{3.2k}$$

Rules (3.1) and (3.2) are syntax-directed, so we assume an inversion lemma for each rule as needed without stating it separately. The following standard lemmas also hold.

The Weakening Lemma establishes that extending a context with unnecessary hypotheses preserves well-formedness and typing.

Lemma 3.2 (Weakening)**.** *All of the following hold:*

1. *If* $\Delta \vdash \tau$ type *then* Δ, t type $\vdash \tau$ type*.*
2. *If* $\Delta\, \Gamma \vdash e : \tau$ *then* Δ, t type $\Gamma \vdash e : \tau$*.*
3. *If* $\Delta\, \Gamma \vdash e : \tau$ *and* $\Delta \vdash \tau'$ type *then* $\Delta\, \Gamma, x : \tau' \vdash e : \tau$*.*

Proof Sketch. For each part, by rule induction on the assumption. □

We assume that renaming of bound variables, α-equivalence and substitution are defined as in *PFPL* [38]. 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 3.3 (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. *If* Δ, t type $\Gamma \vdash e : \tau$ *and* $\Delta \vdash \tau'$ type *then* $\Delta\, [\tau'/t]\Gamma \vdash [\tau'/t]e : [\tau'/t]\tau$*.*

3. *If* $\Delta\ \Gamma, x : \tau' \vdash e : \tau$ *and* $\Delta\ \Gamma \vdash e' : \tau'$ *then* $\Delta\ \Gamma \vdash [e'/x]e : \tau$.

Proof Sketch. For each part, by rule induction on the first assumption. □

The Decomposition Lemma is the converse of the Substitution Lemma.

Lemma 3.4 (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. *If* $\Delta\ [\tau'/t]\Gamma \vdash [\tau'/t]e : [\tau'/t]\tau$ *and* $\Delta \vdash \tau'$ type *then* Δ, t type $\Gamma \vdash e : \tau$.
3. *If* $\Delta\ \Gamma \vdash [e'/x]e : \tau$ *and* $\Delta\ \Gamma \vdash e' : \tau'$ *then* $\Delta\ \Gamma, x : \tau' \vdash e : \tau$.

Proof Sketch.

1. By rule induction over Rules (3.1) and case analysis on 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 rule induction over Rules (3.2) and case analysis on the definition of substitution. In all cases, the derivation of $\Delta\ [\tau'/t]\Gamma \vdash [\tau'/t]e : [\tau'/t]\tau$ does not depend on the form of τ'.
3. By rule induction over Rules (3.2) and case analysis on the definition of substitution. In all cases, the derivation of $\Delta\ \Gamma \vdash [e'/x]e : \tau$ does not depend on the form of e'.

□

The Regularity Lemma establishes that the type assigned to an expanded expression under a well-formed typing context is always well-formed.

Lemma 3.5 (Regularity)**.** *If* $\Delta\ \Gamma \vdash e : \tau$ *and* $\Delta \vdash \Gamma$ ctx *then* $\Delta \vdash \tau$ type.

Proof Sketch. By rule induction over Rules (3.2) and application of Definition 3.1 and Lemma 3.3. □

3.2.3 Structural Dynamics

The *structural dynamics of* $\mathsf{miniVerse}_{\mathbf{UE}}$ is specified 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 3.6 (Iterated Transition)**.** $e \mapsto^* e'$ *is the reflexive, transitive closure of* $e \mapsto e'$.

Definition 3.7 (Evaluation)**.** $e \Downarrow e'$ *iff* $e \mapsto^* e'$ *and* e' val.

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

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

Condition 3.8 (Canonical Forms)**.** *If* $\vdash e : \tau$ *and* e val *then:*

1. *If* $\tau =$ `parr(`τ_1`;`τ_2`)` *then* $e =$ `lam{`τ_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_ℓ, *we have that* $L = L', \ell$ *and* $\tau = \texttt{sum}[L', \ell](\{i \hookrightarrow \tau_i\}_{i \in L'}; \ell \hookrightarrow \tau_\ell)$ *and* $e = \texttt{in}[L', \ell][\ell]\{\{i \hookrightarrow \tau_i\}_{i \in L'}; \ell \hookrightarrow \tau_\ell\}(e')$ *and* $\vdash e' : \tau_\ell$ *and* e' val.

The Preservation condition ensures that evaluation preserves typing.

Condition 3.9 (Preservation)**.** *If* $\vdash e : \tau$ *and* $e \mapsto^* e'$ *then* $\vdash e' : \tau$.

The Progress condition ensures that evaluation of a well-typed expanded expression cannot "get stuck".

Condition 3.10 (Progress)**.** *If* $\vdash e : \tau$ *then either* e val *or there exists an* e' *such that* $e \mapsto e'$.

Together, these two conditions constitute the Type Safety Condition.

3.2.4 Syntax of the Outer Surface

A miniVerse$_{\mathbf{UE}}$ 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 parseUExp(b) to produce an *unexpanded expression*, $\hat{e}$. Unexpanded expressions can contain *unexpanded types*, $\hat{\tau}$, so we also need a partial metafunction parseUTyp(b). The abstract syntax of unexpanded types and expressions, which form the *outer surface* of miniVerse$_{\mathbf{UE}}$, is defined in Figure 3.5. The full definition of the textual syntax of miniVerse$_{\mathbf{UE}}$, which parseUExp(b) and parseUTyp(b) implement, 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 and expression.

Condition 3.11 (Textual Representability)**.** *Both of the following must hold:*

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

Unexpanded types and expressions are given meaning by expansion to types and expanded expressions, respectively, according to the *typed expansion judgements*, which are defined in the next subsection.

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 expansion. We **cannot** adopt the usual definitions 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 3.4) maps onto an outer surface form. We refer to these as the *shared forms*. In particular:

- Each type variable, t, maps onto a unique type sigil, written $\widehat{t}$ (pronounced "sigil of t"). Notice the distinction between $\hat{t}$, which is a metavariable ranging over type

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
		$\texttt{ulam}\{\hat{\tau}\}(\hat{x}.\hat{e})$	$\lambda \hat{x}{:}\hat{\tau}.\hat{e}$	abstraction
		$\texttt{uap}(\hat{e};\hat{e})$	$\hat{e}(\hat{e})$	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
		$\texttt{ufold}\{\hat{t}.\hat{\tau}\}(\hat{e})$	$\texttt{fold}(\hat{e})$	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
		$\texttt{upr}[\ell](\hat{e})$	$\hat{e} \cdot \ell$	projection
		$\texttt{uin}[L][\ell]\{\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}\}(\hat{e})$	$\texttt{inj}\ \ell \cdot \hat{e}$	injection
		$\texttt{ucase}[L]\{\hat{\tau}\}(\hat{e};\{i \hookrightarrow \hat{x}_i.\hat{e}_i\}_{i \in L})$	$\texttt{case}\ \hat{e}\ \{i \hookrightarrow \hat{x}_i.\hat{e}_i\}_{i \in L}$	case analysis
		$\texttt{usyntaxue}\{e\}\{\hat{\tau}\}(\hat{a}.\hat{e})$	$\texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \{e\}\ \texttt{in}\ \hat{e}$	ueTSM definition
		$\texttt{uapuetsm}[b][\hat{a}]$	$\hat{a}\ /b/$	ueTSM application

Figure 3.5: Abstract syntax of unexpanded types and expressions in $\mathsf{miniVerse}_{\mathbf{UE}}$. Metavariable $\hat{t}$ ranges over type sigils, $\hat{x}$ ranges over expression sigils, $\hat{a}$ over TSM names and b over textual sequences, which, when they appear in an unexpanded expression, are called literal bodies. Literal bodies might contain spliced subterms that are only "surfaced" during typed expansion, so renaming of bound identifiers and substitution are not defined over unexpanded types and expressions.

sigils, and $\widehat{t}$, which is a metafunction, written in stylized form, applied to a type variable to produce a type sigil.

- Each type form, τ, maps onto an unexpanded type form, $\mathcal{U}(\tau)$, as follows:

$$\begin{aligned}
\mathcal{U}(t) &= \widehat{t} \\
\mathcal{U}(\texttt{parr}(\tau_1;\tau_2)) &= \texttt{uparr}(\mathcal{U}(\tau_1);\mathcal{U}(\tau_2)) \\
\mathcal{U}(\texttt{all}(t.\tau)) &= \texttt{uall}(\widehat{t}.\mathcal{U}(\tau)) \\
\mathcal{U}(\texttt{rec}(t.\tau)) &= \texttt{urec}(\widehat{t}.\mathcal{U}(\tau)) \\
\mathcal{U}(\texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})) &= \texttt{uprod}[L](\{i \hookrightarrow \mathcal{U}(\tau_i)\}_{i\in L}) \\
\mathcal{U}(\texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i\in L})) &= \texttt{usum}[L](\{i \hookrightarrow \mathcal{U}(\tau_i)\}_{i\in L})
\end{aligned}$$

- Each expression variable, x, maps onto a unique expression sigil written $\widehat{x}$. Again, notice the distinction between $\hat{x}$ and $\widehat{x}$.
- Each expanded expression form, e, maps onto an unexpanded expression form, $\mathcal{U}(e)$, as follows:

$$\begin{aligned}
\mathcal{U}(x) &= \widehat{x} \\
\mathcal{U}(\texttt{lam\{}\tau\texttt{\}(}x.e\texttt{)}) &= \texttt{ulam\{}\mathcal{U}(\tau)\texttt{\}(}\widehat{x}.\mathcal{U}(e)\texttt{)} \\
\mathcal{U}(\texttt{ap(}e_1;e_2\texttt{)}) &= \texttt{uap(}\mathcal{U}(e_1);\mathcal{U}(e_2)\texttt{)} \\
\mathcal{U}(\texttt{tlam(}t.e\texttt{)}) &= \texttt{utlam(}\widehat{t}.\mathcal{U}(e)\texttt{)} \\
\mathcal{U}(\texttt{tap\{}\tau\texttt{\}(}e\texttt{)}) &= \texttt{utap\{}\mathcal{U}(\tau)\texttt{\}(}\mathcal{U}(e)\texttt{)} \\
\mathcal{U}(\texttt{fold\{}t.\tau\texttt{\}(}e\texttt{)}) &= \texttt{ufold\{}\widehat{t}.\mathcal{U}(\tau)\texttt{\}(}\mathcal{U}(e)\texttt{)} \\
\mathcal{U}(\texttt{unfold(}e\texttt{)}) &= \texttt{uunfold(}\mathcal{U}(e)\texttt{)} \\
\mathcal{U}(\texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L})) &= \texttt{utpl}[L](\{i \hookrightarrow \mathcal{U}(e_i)\}_{i\in L}) \\
\mathcal{U}(\texttt{in}[L][\ell]\{\{i \hookrightarrow \tau_i\}_{i\in L}\}(e)) &= \texttt{uin}[L][\ell]\{\{i \hookrightarrow \mathcal{U}(\tau_i)\}_{i\in L}\}(\mathcal{U}(e)) \\
\mathcal{U}(\texttt{case}[L]\{\tau\}(e;\{i \hookrightarrow x_i.e_i\}_{i\in L})) &= \texttt{ucase}[L]\{\mathcal{U}(\tau)\}(\mathcal{U}(e);\{i \hookrightarrow \widehat{x}_i.\mathcal{U}(e_i)\}_{i\in L})
\end{aligned}$$

There are only two unexpanded expression forms, highlighted in gray in Figure 3.5, that do not correspond to expanded expression forms – the ueTSM definition form and the ueTSM application form.

3.2.5 Typed Expansion

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

Judgement Form	**Description**
$\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau$ type	$\hat{\tau}$ is well-formed and has expansion τ assuming $\hat{\Delta}$
$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau$	$\hat{e}$ has expansion e and type τ under ueTSM context $\hat{\Psi}$ assuming $\hat{\Delta}$ and $\hat{\Gamma}$

Type Expansion

The *type expansion judgement*, $\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \ \mathsf{type}$, is inductively defined by the following rules.

$$\frac{}{\hat{\Delta}, \hat{t} \rightsquigarrow t \ \mathsf{type} \vdash \hat{t} \rightsquigarrow t \ \mathsf{type}} \tag{3.3a}$$

$$\frac{\hat{\Delta} \vdash \hat{\tau}_1 \rightsquigarrow \tau_1 \ \mathsf{type} \qquad \hat{\Delta} \vdash \hat{\tau}_2 \rightsquigarrow \tau_2 \ \mathsf{type}}{\hat{\Delta} \vdash \mathtt{uparr}(\hat{\tau}_1; \hat{\tau}_2) \rightsquigarrow \mathtt{parr}(\tau_1; \tau_2) \ \mathsf{type}} \tag{3.3b}$$

$$\frac{\hat{\Delta}, \hat{t} \rightsquigarrow t \ \mathsf{type} \vdash \hat{\tau} \rightsquigarrow \tau \ \mathsf{type}}{\hat{\Delta} \vdash \mathtt{uall}(\hat{t}.\hat{\tau}) \rightsquigarrow \mathtt{all}(t.\tau) \ \mathsf{type}} \tag{3.3c}$$

$$\frac{\hat{\Delta}, \hat{t} \rightsquigarrow t \ \mathsf{type} \vdash \hat{\tau} \rightsquigarrow \tau \ \mathsf{type}}{\hat{\Delta} \vdash \mathtt{urec}(\hat{t}.\hat{\tau}) \rightsquigarrow \mathtt{rec}(t.\tau) \ \mathsf{type}} \tag{3.3d}$$

$$\frac{\{\hat{\Delta} \vdash \hat{\tau}_i \rightsquigarrow \tau_i \ \mathsf{type}\}_{i \in L}}{\hat{\Delta} \vdash \mathtt{uprod}[L](\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}) \rightsquigarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \ \mathsf{type}} \tag{3.3e}$$

$$\frac{\{\hat{\Delta} \vdash \hat{\tau}_i \rightsquigarrow \tau_i \ \mathsf{type}\}_{i \in L}}{\hat{\Delta} \vdash \mathtt{usum}[L](\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}) \rightsquigarrow \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \ \mathsf{type}} \tag{3.3f}$$

Unexpanded type formation contexts, $\hat{\Delta}$, are of the form $\langle \mathcal{D}; \Delta \rangle$, where $\mathcal{D}$ is a *type sigil expansion context*, and Δ is a type formation context. A type sigil expansion context, $\mathcal{D}$, is a finite function that maps each type sigil $\hat{t} \in \mathrm{dom}(\mathcal{D})$ to the hypothesis $\hat{t} \rightsquigarrow t$, for some type variable t. We write $\mathcal{D} \uplus \hat{t} \rightsquigarrow t$ for the type sigil expansion context that maps $\hat{t}$ to $\hat{t} \rightsquigarrow t$ and defers to $\mathcal{D}$ for all other type sigils (i.e. the previous mapping, if it exists, is updated). We define $\hat{\Delta}, \hat{t} \rightsquigarrow t \ \mathsf{type}$ when $\hat{\Delta} = \langle \mathcal{D}; \Delta \rangle$ as an abbreviation of

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

To understand how type sigil expansion contexts operate, it is instructive to derive an expansion for the unexpanded type $\forall \hat{t}.\forall \hat{t}.\hat{t}$, or in operational form, $\mathtt{uall}(\hat{t}.\mathtt{uall}(\hat{t}.\hat{t}))$:

$$\dfrac{\dfrac{\dfrac{}{\langle \hat{t} \rightsquigarrow t'; t \ \mathsf{type}, t' \ \mathsf{type} \rangle \vdash \hat{t} \rightsquigarrow t' \ \mathsf{type}} \text{ (3.3a)}}{\langle \hat{t} \rightsquigarrow t; t \ \mathsf{type} \rangle \vdash \mathtt{uall}(\hat{t}.\hat{t}) \rightsquigarrow \mathtt{all}(t'.t') \ \mathsf{type}} \text{ (3.3c)}}{\langle \varnothing; \varnothing \rangle \vdash \mathtt{uall}(\hat{t}.\mathtt{uall}(\hat{t}.\hat{t})) \rightsquigarrow \mathtt{all}(t.\mathtt{all}(t'.t')) \ \mathsf{type}} \text{ (3.3c)}$$

Notice that when a type sigil is bound, a fresh type variable is generated. The type sigil expansion context is extended (when the outermost binding is encountered) or updated (at all inner bindings) and the type formation context is simultaneously extended at each binding (so that typing contexts and ueTSM contexts, discussed below, that contain types that refer to the previous binding remain well-formed). Had we used type variables in the syntax and type formation contexts in the rules above, rather than type sigils and type sigil expansion contexts, derivations for unexpanded types where an

inner binding shadows an outer binding 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.

These rules validate the following lemmas. The Type Expansion Lemma establishes that the expansion of an unexpanded type is a well-formed type.

Lemma 3.12 (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 (3.3). In each case, we apply the IH to or over each premise, then apply the corresponding type formation rule in Rules (3.1). □

The Type Expressibility Lemma establishes that every well-formed type, τ, can be expressed as a well-formed unexpanded type, $\mathcal{U}(\tau)$. This requires defining the meta-function $\mathcal{U}(\Delta)$ which maps Δ onto an unexpanded type formation context as follows:

$$\mathcal{U}(\emptyset) = \langle \emptyset; \emptyset \rangle$$
$$\mathcal{U}(\Delta, t \text{ type}) = \mathcal{U}(\Delta), \widehat{t} \rightsquigarrow t \text{ type}$$

Lemma 3.13 (Type Expressibility)**.** *If* $\Delta \vdash \tau$ type *then* $\mathcal{U}(\Delta) \vdash \mathcal{U}(\tau) \rightsquigarrow \tau$ type.

Proof. By rule induction over Rules (3.1) using the definitions of $\mathcal{U}(\tau)$ and $\mathcal{U}(\Delta)$ above. In each case, we apply the IH to or over each premise, then apply the corresponding type expansion rule in Rules (3.3). □

Typed Expression Expansion

Unexpanded typing contexts, $\hat{\Gamma}$, are of the form $\langle \mathcal{G}; \Gamma \rangle$, where $\mathcal{G}$ is an *expression sigil expansion context*, and Γ is a typing context. An expression sigil expansion context, $\mathcal{G}$, is a finite function that maps each expression sigil $\hat{x} \in \text{dom}(\mathcal{G})$ to the hypothesis $\hat{x} \rightsquigarrow x$, for some expression variable, x. We write $\mathcal{G} \uplus \hat{x} \rightsquigarrow x$ for the expression sigil expansion context that maps $\hat{x}$ to $\hat{x} \rightsquigarrow x$ and defers to $\mathcal{G}$ for all other expression sigils (i.e. the previous mapping, if it exists, 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 (3.4) as follows.

Shared Forms Rules (3.4a) through (3.4k) handle unexpanded expressions of shared form. The first five of these rules are defined below:

$$\frac{}{\hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi}} \hat{x} \rightsquigarrow x : \tau} \tag{3.4a}$$

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \text{ type} \qquad \hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \texttt{ulam\{}\hat{\tau}\texttt{\}(}\hat{x}.\hat{e}\texttt{)} \rightsquigarrow \texttt{lam\{}\tau\texttt{\}(}x.e\texttt{)} : \texttt{parr(}\tau\texttt{;}\tau'\texttt{)}} \tag{3.4b}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e}_1 \rightsquigarrow e_1 : \mathtt{parr}(\tau;\tau') \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e}_2 \rightsquigarrow e_2 : \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \mathtt{uap}(\hat{e}_1;\hat{e}_2) \rightsquigarrow \mathtt{ap}(e_1;e_2) : \tau'} \tag{3.4c}$$

$$\frac{\hat{\Delta}, \hat{t} \rightsquigarrow t\ \mathsf{type}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \mathtt{utlam}(\hat{t}.\hat{e}) \rightsquigarrow \mathtt{tlam}(t.e) : \mathtt{all}(t.\tau)} \tag{3.4d}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \mathtt{all}(t.\tau) \qquad \hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'\ \mathsf{type}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \mathtt{utap}\{\hat{\tau}'\}(\hat{e}) \rightsquigarrow \mathtt{tap}\{\tau'\}(e) : [\tau'/t]\tau} \tag{3.4e}$$

Observe that, in each of these rules, the unexpanded and expanded expression forms in the conclusion correspond, and the premises correspond to those of the typing rule for the expanded expression form, i.e. Rules (3.2a) through (3.2e), respectively. In particular, each type expansion premise in each rule above corresponds to a type formation premise in the corresponding typing rule, and each typed expression expansion premise in each rule above corresponds to a typing premise in 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 ueTSM context, $\hat{\Psi}$, passes opaquely through these rules (we will define ueTSM contexts below). Rules (3.3) were similarly generated by mechanically transforming Rules (3.1).

We can express this scheme more precisely with the following rule transformation. For each rule in Rules (3.1) and Rules (3.2),

$$\frac{J_1 \qquad \cdots \qquad J_k}{J}$$

the corresponding typed expansion rule is

$$\frac{\mathcal{U}(J_1) \qquad \cdots \qquad \mathcal{U}(J_k)}{\mathcal{U}(J)}$$

where

$$\begin{aligned}
\mathcal{U}(\Delta \vdash \tau\ \mathsf{type}) &= \mathcal{U}(\Delta) \vdash \mathcal{U}(\tau) \rightsquigarrow \tau\ \mathsf{type} \\
\mathcal{U}(\Gamma\ \Delta \vdash e : \tau) &= \mathcal{U}(\Gamma)\ \mathcal{U}(\Delta) \vdash_{\hat{\Psi}} \mathcal{U}(e) \rightsquigarrow e : \tau \\
\mathcal{U}(\{J_i\}_{i \in L}) &= \{\mathcal{U}(J_i)\}_{i \in L}
\end{aligned}$$

and where:

- $\mathcal{U}(\tau)$ is defined as follows:
  - When τ is of definite form, $\mathcal{U}(\tau)$ is defined as in Sec. 3.2.4.
  - When τ is of indefinite form, $\mathcal{U}(\tau)$ is a uniquely corresponding metavariable of sort UTyp also of indefinite form. For example, in Rule (3.1b), τ_1 and τ_2 are of indefinite form, i.e. they match arbitrary types. The rule transformation simply "hats" them, i.e. $\mathcal{U}(\tau_1) = \hat{\tau}_1$ and $\mathcal{U}(\tau_2) = \hat{\tau}_2$.
- $\mathcal{U}(e)$ is defined as follows

- When e is of definite form, $\mathcal{U}(e)$ is defined as in Sec. 3.2.4.
- When e is of indefinite form, $\mathcal{U}(e)$ is a uniquely corresponding metavariable of sort UExp also of indefinite form. For example, $\mathcal{U}(e_1) = \hat{e}_1$ and $\mathcal{U}(e_2) = \hat{e}_2$.

- $\mathcal{U}(\Delta)$ is defined as follows:
  - When Δ is of definite form, $\mathcal{U}(\Delta)$ is defined as above.
  - When Δ is of indefinite form, $\mathcal{U}(\Delta)$ is a uniquely corresponding metavariable ranging over unexpanded type formation contexts. For example, $\mathcal{U}(\Delta) = \hat{\Delta}$.
- $\mathcal{U}(\Gamma)$ is defined as follows:
  - When Γ is of definite form, $\mathcal{U}(\Gamma)$ produces the corresponding unexpanded typing context as follows:

$$\begin{aligned}\mathcal{U}(\varnothing) &= \langle \varnothing; \varnothing \rangle \\ \mathcal{U}(\Gamma, x : \tau) &= \mathcal{U}(\Gamma), \widehat{x} \rightsquigarrow x : \tau\end{aligned}$$

  - When Γ is of indefinite form, $\mathcal{U}(\Gamma)$ is a uniquely corresponding metavariable ranging over unexpanded typing contexts. For example, $\mathcal{U}(\Gamma) = \hat{\Gamma}$.

It is instructive to use this rule transformation to generate Rules (3.3) and Rules (3.4a) through (3.4e) above. We omit the remaining rules, i.e. Rules (3.4f) through (3.4k). By instead defining these rules solely by the rule transformation just described, we avoid having to write down a number of rules that are of limited marginal interest. Moreover, this demonstrates the general technique for generating typed expansion rules for unexpanded types and expressions of shared form, so our exposition is somewhat "robust" to changes to the inner core.

We can now establish the Expressibility Theorem – that each well-typed expanded expression, e, can be expressed as an unexpanded expression, $\hat{e}$, and assigned the same type under the corresponding contexts.

Theorem 3.14 (Expressibility). *If* $\Delta\ \Gamma \vdash e : \tau$ *then* $\mathcal{U}(\Delta)\ \mathcal{U}(\Gamma) \vdash_{\hat{\Psi}} \mathcal{U}(e) \rightsquigarrow e : \tau$.

Proof. By rule induction over Rules (3.2). The above rule transformation guarantees that this theorem holds by its construction. In particular, in each case, we can apply Lemma 3.13 to or over each type formation premise, the IH to or over each typing premise, then apply the corresponding rule in Rules (3.4). □

ueTSM Definition and Application The two remaining typed expansion rules, Rules (3.4l) and (3.4m), govern the ueTSM definition and application forms, and are defined in the next two subsections, respectively.

3.2.6 ueTSM Definitions

The stylized ueTSM definition form is

$$\texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \{e_{\text{parse}}\}\ \texttt{in}\ \hat{e}$$

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

The parse function is an expanded expression because parse functions are applied statically (i.e. during typed expansion of $\hat{e}$), as we will discuss when describing ueTSM application below, and evaluation is defined only for closed expanded expressions. This construction simplifies our exposition, though it is not entirely practical because it provides no way for TSM providers to share values between parse functions, nor any way to use TSMs when defining other TSMs. We discuss enriching the language to eliminate these limitations in Sec. 6.2, but it is pedagogically simpler to leave the necessary machinery out of our calculus for now.

Rule (3.4l) defines typed expansion of ueTSM definitions (we use stylized forms for clarity):

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \text{ type} \qquad \varnothing\,\varnothing \vdash e_{\text{parse}} : \texttt{Body} \rightharpoonup \texttt{ParseResultExp} \qquad \hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}})} \hat{e} \rightsquigarrow e : \tau'}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \{e_{\text{parse}}\}\ \texttt{in}\ \hat{e} \rightsquigarrow e : \tau'} \tag{3.4l}$$

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

1. The first premise ensures that the unexpanded type annotation is well-formed and expands it to produce the *type annotation*, τ.
2. The second premise checks that the parse function, e_{parse}, is closed and of type

   $$\texttt{Body} \rightharpoonup \texttt{ParseResultExp}$$

   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* $b \downarrow e_{\text{body}}$. An inverse mapping is defined by the *body decoding judgement* $e_{\text{body}} \uparrow b$.

Judgement Form	**Description**
$b \downarrow e$	b has encoding e
$e \uparrow b$	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 3.15 (Body Isomorphism)**.** *All of the following must hold:*

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

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

   (c) *If $b \downarrow e_{body}$ then $e_{body} \uparrow b$.*

   (d) *If $\vdash e_{body} : \texttt{Body}$ and e_{body} val and $e_{body} \uparrow b$ then $b \downarrow e_{body}$.*

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

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

`ParseResultExp` abbreviates a labeled sum type that distinguishes successful parses from parse errors[1]:

$$\texttt{ParseResultExp} \triangleq [\texttt{Success} \hookrightarrow \texttt{CEExp}, \texttt{ParseError} \hookrightarrow \langle\rangle]$$

The type abbreviated `CEExp` classifies encodings of *candidate expansion expressions* (or *ce-expressions*), $\grave{e}$ (pronounced "grave e"). The syntax of ce-expressions will be described in Sec. 3.2.8. The mapping from ce-expressions to values of type `CEExp` is defined by the *ce-expression encoding judgement*, $\grave{e} \downarrow_{\mathsf{CEExp}} e$. An inverse mapping is defined by the *ce-expression decoding judgement*, $e \uparrow_{\mathsf{CEExp}} \grave{e}$.

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

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

Condition 3.16 (Candidate Expansion Expression Isomorphism)**.** *All of the following must hold:*

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

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

(c) If $\grave{e} \downarrow_{\mathsf{CEExp}} e_{cand}$ then $e_{cand} \uparrow_{\mathsf{CEExp}} \grave{e}$.

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

(e) If $\grave{e} \downarrow_{\mathsf{CEExp}} e_{cand}$ and $\grave{e} \downarrow_{\mathsf{CEExp}} e'_{cand}$ then $e_{cand} = e'_{cand}$.

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

3. The final premise of Rule (3.4l) extends the ueTSM context, $\hat{\Psi}$, with the newly determined ueTSM definition, and proceeds to assign a type, τ', and expansion, e, to $\hat{e}$. The conclusion of Rule (3.4l) assigns this type and expansion to the ueTSM definition as a whole.

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

   A *TSM naming context*, $\mathcal{A}$, is a finite function mapping each TSM name $\hat{a} \in \mathrm{dom}(\mathcal{A})$ to the *TSM name-symbol mapping*, $\hat{a} \rightsquigarrow a$, for some *symbol*, a. We write $\mathcal{A} \uplus \hat{a} \rightsquigarrow a$ for the ueTSM naming context that maps $\hat{a}$ to $\hat{a} \rightsquigarrow a$, and defers to $\mathcal{A}$ for all other TSM names (i.e. the previous mapping, if it exists, is updated).

   A *ueTSM definition context*, Ψ, is a finite function mapping each symbol $a \in \mathrm{dom}(\Psi)$ to an *expanded ueTSM definition*, $a \hookrightarrow \texttt{uetsm}(\tau; e_{\mathrm{parse}})$, where τ is the ueTSM's type

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

annotation, and e_{parse} is its parse function. We write $\Psi, a \hookrightarrow \texttt{uetsm}(\tau;\, e_{\text{parse}})$ when $a \notin \text{dom}(\Psi)$ for the extension of Ψ that maps a to $a \hookrightarrow \texttt{uetsm}(\tau;\, e_{\text{parse}})$. We write $\Delta \vdash \Psi$ ueTSMs when all the type annotations in Ψ are well-formed assuming Δ, and the parse functions in Ψ are closed and of type $\texttt{Body} \rightharpoonup \texttt{ParseResultExp}$.

Definition 3.17 (ueTSM Definition Context Formation)**.** $\Delta \vdash \Psi$ ueTSMs *iff for each* $\hat{a} \hookrightarrow \texttt{uetsm}(\tau;\, e_{parse}) \in \Psi$, *we have* $\Delta \vdash \tau$ type *and* $\varnothing\ \varnothing \vdash e_{parse} : \texttt{Body} \rightharpoonup \texttt{ParseResultExp}$.

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

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

The reason we distinguish TSM names, $\hat{a}$, from symbols, a, is in order to support TSM abbreviations, which we defer discussion of until Chapter 5.

3.2.7 ueTSM Application

The stylized unexpanded expression form for applying a ueTSM 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 stylized variants of any of the literal forms specified in Figure 3.1 could also be added to Figure 3.5. The corresponding operational form is $\texttt{uapuetsm}[b]\texttt{[}\hat{a}\texttt{]}$.

The typed expansion rule governing ueTSM application is below:

$$\frac{\begin{array}{c} b \downarrow e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj Success} \cdot e_{\text{cand}} \qquad e_{\text{cand}} \uparrow_{\texttt{CEExp}} \grave{e} \\ \varnothing\ \varnothing \vdash^{\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}}); b} \grave{e} \rightsquigarrow e : \tau \end{array}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}})} \hat{a}\ /b/ \rightsquigarrow e : \tau} \tag{3.4m}$$

The premises of Rule (3.4m) can be understood as follows, in order:

1. The first premise determines the encoding of the literal body, e_{body} (see above).
2. The second premise applies the parse function e_{parse}, which appears in the ueTSM context associated with $\hat{a}$, to e_{body}. If parsing succeeds, i.e. a value of the (stylized) form $\texttt{inj Success} \cdot e_{\text{cand}}$ results from evaluation, then e_{cand} will be a value of type CEExp (assuming a well-formed ueTSM context, by application of Assumption 3.9). We call e_{cand} the *encoding of the candidate expansion*.

   If the parse function produces a value labeled ParseError, then typed expansion fails. No rule is necessary to handle this case.
3. The third premise decodes the encoding of the candidate expansion to produce the *candidate expansion*, $\grave{e}$ (see above).
4. The final premise of Rule (3.4m) *validates* the candidate expansion and simultaneously generates the *final expansion*, e. This is the topic of Sec. 3.2.9.

Sort	Operational Form	Stylized Form	Description
CETyp $\grave{\tau}$ ::=	t	t	variable
	$\texttt{ceparr}(\grave{\tau};\grave{\tau})$	$\grave{\tau} \rightharpoonup \grave{\tau}$	partial function
	$\texttt{ceall}(t.\grave{\tau})$	$\forall t.\grave{\tau}$	polymorphic
	$\texttt{cerec}(t.\grave{\tau})$	$\mu t.\grave{\tau}$	recursive
	$\texttt{ceprod}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i\in L})$	$\langle\{i \hookrightarrow \grave{\tau}_i\}_{i\in L}\rangle$	labeled product
	$\texttt{cesum}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i\in L})$	$[\{i \hookrightarrow \grave{\tau}_i\}_{i\in L}]$	labeled sum
	$\texttt{cesplicedt}[m;n]$	$\texttt{splicedt}\langle m,n\rangle$	spliced
CEExp $\grave{e}$ ::=	x	x	variable
	$\texttt{celam}\{\grave{\tau}\}(x.\grave{e})$	$\lambda x{:}\grave{\tau}.\grave{e}$	abstraction
	$\texttt{ceap}(\grave{e};\grave{e})$	$\grave{e}(\grave{e})$	application
	$\texttt{cetlam}(t.\grave{e})$	$\Lambda t.\grave{e}$	type abstraction
	$\texttt{cetap}\{\grave{\tau}\}(\grave{e})$	$\grave{e}[\grave{\tau}]$	type application
	$\texttt{cefold}\{t.\grave{\tau}\}(\grave{e})$	$\texttt{fold}(\grave{e})$	fold
	$\texttt{ceunfold}(\grave{e})$	$\texttt{unfold}(\grave{e})$	unfold
	$\texttt{cetpl}[L](\{i \hookrightarrow \grave{e}_i\}_{i\in L})$	$\langle\{i \hookrightarrow \grave{e}_i\}_{i\in L}\rangle$	labeled tuple
	$\texttt{cepr}[\ell](\grave{e})$	$\grave{e} \cdot \ell$	projection
	$\texttt{cein}[L][\ell]\{\{i \hookrightarrow \grave{\tau}_i\}_{i\in L}\}(\grave{e})$	$\texttt{inj}\ \ell \cdot \grave{e}$	injection
	$\texttt{cecase}[L]\{\tau\}(\grave{e};\{i \hookrightarrow x_i.\grave{e}_i\}_{i\in L})$	$\texttt{case}\ \grave{e}\ \{i \hookrightarrow x_i.\grave{e}_i\}_{i\in L}$	case analysis
	$\texttt{cesplicede}[m;n]$	$\texttt{splicede}\langle m,n\rangle$	spliced

Figure 3.6: Abstract syntax of candidate expansion types and expressions in $\mathsf{miniVerse}_{\mathbf{UE}}$. Metavariables m and n range over natural numbers. Candidate expansion types and expressions are identified up to α-equivalence.

3.2.8 Syntax of Candidate Expansions

Figure 3.6 defines the syntax of candidate expansion types (or *ce-types*), $\grave{\tau}$, and candidate expansion expressions (or *ce-expressions*), $\grave{e}$. Candidate expansion types and expressions are identified up to α-equivalence in the usual manner.

Each inner core form maps onto a candidate expansion form. We refer to these as the *shared forms*. In particular:

- Each type form maps onto a ce-type form according to the metafunction $\mathcal{C}(\tau)$, defined as follows:

$$\begin{aligned}
\mathcal{C}(t) &= t \\
\mathcal{C}(\texttt{parr}(\tau_1;\tau_2)) &= \texttt{ceparr}(\mathcal{C}(\tau_1);\mathcal{C}(\tau_2)) \\
\mathcal{C}(\texttt{all}(t.\tau)) &= \texttt{ceall}(t.\mathcal{C}(\tau)) \\
\mathcal{C}(\texttt{rec}(t.\tau)) &= \texttt{cerec}(t.\mathcal{C}(\tau)) \\
\mathcal{C}(\texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})) &= \texttt{ceprod}[L](\{i \hookrightarrow \mathcal{C}(\grave{\tau}_i)\}_{i\in L}) \\
\mathcal{C}(\texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i\in L})) &= \texttt{cesum}[L](\{i \hookrightarrow \mathcal{C}(\grave{\tau}_i)\}_{i\in L})
\end{aligned}$$

- Each expanded expression form maps onto a ce-expression form according to the

metafunction $\mathcal{C}(e)$, defined as follows:

$$\begin{aligned}
\mathcal{C}(x) &= x \\
\mathcal{C}(\texttt{lam\{}\tau\texttt{\}(}x.e\texttt{)}) &= \texttt{celam\{}\mathcal{C}(\tau)\texttt{\}(}x.\mathcal{C}(e)\texttt{)} \\
\mathcal{C}(\texttt{ap(}e_1\texttt{;}e_2\texttt{)}) &= \texttt{ceap(}\mathcal{C}(e_1)\texttt{;}\mathcal{C}(e_2)\texttt{)} \\
\mathcal{C}(\texttt{tlam(}t.e\texttt{)}) &= \texttt{cetlam(}t.\mathcal{C}(e)\texttt{)} \\
\mathcal{C}(\texttt{tap\{}\tau\texttt{\}(}e\texttt{)}) &= \texttt{cetap\{}\mathcal{C}(\tau)\texttt{\}(}\mathcal{C}(e)\texttt{)} \\
\mathcal{C}(\texttt{fold\{}t.\tau\texttt{\}(}e\texttt{)}) &= \texttt{cefold\{}t.\mathcal{C}(\tau)\texttt{\}(}\mathcal{C}(e)\texttt{)} \\
\mathcal{C}(\texttt{unfold(}e\texttt{)}) &= \texttt{ceunfold(}\mathcal{C}(e)\texttt{)} \\
\mathcal{C}(\texttt{tpl}[L]\texttt{(\{}i \hookrightarrow e_i\texttt{\}}_{i\in L}\texttt{)}) &= \texttt{cetpl}[L]\texttt{(\{}i \hookrightarrow \mathcal{C}(e_i)\texttt{\}}_{i\in L}\texttt{)} \\
\mathcal{C}(\texttt{in}[L]\texttt{[}\ell\texttt{]\{\{}i \hookrightarrow \tau_i\texttt{\}}_{i\in L}\texttt{\}(}e\texttt{)}) &= \texttt{cein}[L]\texttt{[}\ell\texttt{]\{\{}i \hookrightarrow \mathcal{C}(\tau_i)\texttt{\}}_{i\in L}\texttt{\}(}\mathcal{C}(e)\texttt{)} \\
\mathcal{C}(\texttt{case}[L]\texttt{\{}\tau\texttt{\}(}e\texttt{;\{}i \hookrightarrow x_i.e_i\texttt{\}}_{i\in L}\texttt{)}) &= \texttt{cecase}[L]\texttt{\{}\mathcal{C}(\tau)\texttt{\}(}\mathcal{C}(e)\texttt{;\{}i \hookrightarrow x_i.\mathcal{C}(e_i)\texttt{\}}_{i\in L}\texttt{)}
\end{aligned}$$

There are two other candidate expansion forms, highlighted in gray in Figure 3.6: a ce-type form for *references to spliced unexpanded types*, $\texttt{cesplicedt}[m;n]$, and a ce-expression form for *references to spliced unexpanded expressions*, $\texttt{cesplicede}[m;n]$.

3.2.9 Candidate Expansion Validation

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

Judgement Form	Description
$\Delta \vdash^{\mathbb{T}} \grave{\tau} \rightsquigarrow \tau$ type	Candidate expansion type $\grave{\tau}$ is well-formed and has expansion τ assuming Δ and type splicing scene $\mathbb{T}$.
$\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau$	Candidate expansion expression $\grave{e}$ has expansion e and type τ assuming Δ and Γ and expression splicing scene $\mathbb{E}$.

Expression splicing scenes, $\mathbb{E}$, are of the form $\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; b$, and *type splicing scenes*, $\mathbb{T}$, are of the form $\Delta; b$. We write $\mathsf{ts}(\mathbb{E})$ for the type splicing scene constructed by dropping the unexpanded typing context and ueTSM 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 candidate expansion validation process, the unexpanded type formation context, $\hat{\Delta}$, unexpanded typing context, $\hat{\Gamma}$, ueTSM context, $\hat{\Psi}$, and the literal body, b, from the ueTSM application site (cf. Rule (3.4m)), because these are necessary to validate references to spliced unexpanded types and expressions that appear within a candidate expansion.

Candidate Expansion Type Validation

The *candidate expansion type validation judgement*, $\Delta \vdash^{\mathbb{T}} \grave{\tau} \rightsquigarrow \tau$ type, is inductively defined by Rules (3.5) as follows.

Shared Forms Rules (3.5a) through (3.5f), which validate ce-types of shared form, are defined below.

$$\frac{}{\Delta, t\ \mathsf{type} \vdash^{\mathbb{T}} t \rightsquigarrow t\ \mathsf{type}} \tag{3.5a}$$

$$\frac{\Delta \vdash^{\mathbb{T}} \dot{\tau}_1 \rightsquigarrow \tau_1\ \mathsf{type} \qquad \Delta \vdash^{\mathbb{T}} \dot{\tau}_2 \rightsquigarrow \tau_2\ \mathsf{type}}{\Delta \vdash^{\mathbb{T}} \mathtt{ceparr}(\dot{\tau}_1; \dot{\tau}_2) \rightsquigarrow \mathtt{parr}(\tau_1; \tau_2)\ \mathsf{type}} \tag{3.5b}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash^{\mathbb{T}} \dot{\tau} \rightsquigarrow \tau\ \mathsf{type}}{\Delta \vdash^{\mathbb{T}} \mathtt{ceall}(t.\dot{\tau}) \rightsquigarrow \mathtt{all}(t.\tau)\ \mathsf{type}} \tag{3.5c}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash^{\mathbb{T}} \dot{\tau} \rightsquigarrow \tau\ \mathsf{type}}{\Delta \vdash^{\mathbb{T}} \mathtt{cerec}(t.\dot{\tau}) \rightsquigarrow \mathtt{rec}(t.\tau)\ \mathsf{type}} \tag{3.5d}$$

$$\frac{\{\Delta \vdash^{\mathbb{T}} \dot{\tau}_i \rightsquigarrow \tau_i\ \mathsf{type}\}_{i \in L}}{\Delta \vdash^{\mathbb{T}} \mathtt{ceprod}[L](\{i \hookrightarrow \dot{\tau}_i\}_{i \in L}) \rightsquigarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})\ \mathsf{type}} \tag{3.5e}$$

$$\frac{\{\Delta \vdash^{\mathbb{T}} \dot{\tau}_i \rightsquigarrow \tau_i\ \mathsf{type}\}_{i \in L}}{\Delta \vdash^{\mathbb{T}} \mathtt{cesum}[L](\{i \hookrightarrow \dot{\tau}_i\}_{i \in L}) \rightsquigarrow \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L})\ \mathsf{type}} \tag{3.5f}$$

Observe that, in each of these rules, the ce-type form and the type form in the conclusion correspond, and the premises correspond to those of the corresponding type formation rule, i.e. Rules (3.1). The type splicing scene, $\mathbb{T}$, passes opaquely through these rules. The following lemma establishes that each type can be expressed as a well-formed ce-type, under the same type formation context and any type splicing scene.

Lemma 3.18 (Candidate Expansion Type Expressibility)**.** *If* $\Delta \vdash \tau\ \mathsf{type}$ *then* $\Delta \vdash^{\mathbb{T}} \mathcal{C}(\tau) \rightsquigarrow \tau\ \mathsf{type}$.

Proof. By rule induction over Rules (3.1). In each case, we apply the IH on or over each premise, then apply the corresponding ce-type validation rule in Rules (3.5). □

Notice that in Rule (3.5a), only type variables tracked by the candidate expansion type formation context, Δ, are validated. Type variables in the application site unexpanded type formation context, which appears within the type splicing scene, $\mathbb{T}$, are not validated. Indeed, $\mathbb{T}$ is not inspected by any of the rules above. This achieves *context-independent expansion* as described in Sec. 3.1.3 for type variables – ueTSMs cannot impose "hidden constraints" on the application site unexpanded type formation context, because the type variables bound at the application site are simply not directly available to ce-types.

References to Spliced Types The only ce-type form that does not correspond to a type form is $\mathtt{cesplicedt}[m; n]$, 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, beginning at position m and ending at position n.

Rule (3.5g) governs this form:

$$\frac{\mathsf{parseUTyp}(\mathsf{subseq}(b;m;n)) = \hat{\tau} \qquad \langle \mathcal{D}; \Delta_{\mathrm{app}} \rangle \vdash \hat{\tau} \rightsquigarrow \tau \ \mathsf{type} \qquad \Delta \cap \Delta_{\mathrm{app}} = \emptyset}{\Delta \vdash^{\langle \mathcal{D}; \Delta_{\mathrm{app}} \rangle; b} \mathtt{cesplicedt}[m;n] \rightsquigarrow \tau \ \mathsf{type}} \tag{3.5g}$$

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 $\mathsf{parseUTyp}(b)$, described in Sec. 3.2.4, to produce the spliced unexpanded type itself, $\hat{\tau}$.

The second premise of Rule (3.5g) performs type expansion of $\hat{\tau}$ under the application site unexpanded type formation context, $\langle \mathcal{D}; \Delta_{\mathrm{app}} \rangle$, which appears in the type splicing scene. The hypotheses in the candidate expansion type formation context, Δ, are not made available to τ.

The third premise of Rule (3.5g) imposes the constraint that the candidate 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 candidate expansion that the reference to the spliced type appears within.

This achieves *expansion-independent splicing* as described in Sec. 3.1.3 for type variables – the TSM provider can choose type variable names freely within a candidate expansion, because the language prevents them from shadowing type variables at the application site (by α-varying the candidate expansion as needed).

Rules (3.5) validate the following lemma, which establishes that the final expansion of a valid ce-type is a well-formed type under the combined type formation context.

Lemma 3.19 (Candidate Expansion Type Validation)**.** *If* $\Delta \vdash^{\langle \mathcal{D}; \Delta_{app} \rangle; b} \grave{\tau} \rightsquigarrow \tau \ \mathsf{type}$ *then* $\Delta \cup \Delta_{app} \vdash \tau \ \mathsf{type}$.

Proof. By rule induction over Rules (3.5).

Case (3.5a)**.** We have

(1)	$\Delta = \Delta', t \ \mathsf{type}$	by assumption
(2)	$\grave{\tau} = t$	by assumption
(3)	$\tau = t$	by assumption
(4)	$\Delta', t \ \mathsf{type} \vdash t \ \mathsf{type}$	by Rule (3.1a)
(5)	$\Delta', t \ \mathsf{type} \cup \Delta_{\mathrm{app}} \vdash t \ \mathsf{type}$	by Lemma 3.2 over Δ_{app} to (4)

Case (3.5b)**.** We have

(1)	$\grave{\tau} = \mathtt{ceparr}(\grave{\tau}_1; \grave{\tau}_2)$	by assumption
(2)	$\tau = \mathtt{parr}(\tau_1; \tau_2)$	by assumption
(3)	$\Delta \vdash^{\langle \mathcal{D}; \Delta_{\mathrm{app}} \rangle; b} \grave{\tau}_1 \rightsquigarrow \tau_1 \ \mathsf{type}$	by assumption
(4)	$\Delta \vdash^{\langle \mathcal{D}; \Delta_{\mathrm{app}} \rangle; b} \grave{\tau}_2 \rightsquigarrow \tau_2 \ \mathsf{type}$	by assumption
(5)	$\Delta \cup \Delta_{\mathrm{app}} \vdash \tau_1 \ \mathsf{type}$	by IH on (3)
(6)	$\Delta \cup \Delta_{\mathrm{app}} \vdash \tau_2 \ \mathsf{type}$	by IH on (4)

(7) $\Delta \cup \Delta_{\text{app}} \vdash \texttt{parr}(\tau_1; \tau_2)$ type — by Rule (3.1b) on (5) and (6)

Case (3.5c). We have

(1) $\dot{\tau} = \texttt{ceall}(t.\dot{\tau}')$ — 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 (3.1c) on (5)

Case (3.5d) through (3.5f). These cases follow analagously, i.e. we apply the IH to or over all ce-type validation premises, apply exchange as needed, and then apply the corresponding type formation rule.

Case (3.5g). We have

(1) $\dot{\tau} = \texttt{cesplicedt}[m; n]$ — 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) $\Delta_{\text{app}} \vdash \tau$ type — by Lemma 3.12 on (3)
(5) $\Delta \cup \Delta_{\text{app}} \vdash \tau$ type — by Lemma 3.2 over Δ on (4) and exchange over Δ

□

Candidate Expansion Expression Validation

The *candidate expansion expression validation judgement*, $\Delta\ \Gamma \vdash^{\mathbb{E}} \dot{e} \rightsquigarrow e : \tau$, is defined mutually inductively with the typed expansion judgement by Rules (3.6) as follows.

Shared Forms Rules (3.6a) through (3.6k) validate ce-expressions of shared form. The first three of these rules are defined below:

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash^{\mathbb{E}} x \rightsquigarrow x : \tau} \tag{3.6a}$$

$$\frac{\Delta \vdash^{\mathsf{ts}(\mathbb{E})} \dot{\tau} \rightsquigarrow \tau \text{ type} \qquad \Delta\ \Gamma, x : \tau \vdash^{\mathbb{E}} \dot{e} \rightsquigarrow e : \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \texttt{celam\{}\dot{\tau}\texttt{\}}(x.\dot{e}) \rightsquigarrow \texttt{lam\{}\tau\texttt{\}}(x.e) : \texttt{parr}(\tau; \tau')} \tag{3.6b}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \dot{e}_1 \rightsquigarrow e_1 : \texttt{parr}(\tau; \tau') \qquad \Delta\ \Gamma \vdash^{\mathbb{E}} \dot{e}_2 \rightsquigarrow e_2 : \tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \texttt{ceap}(\dot{e}_1; \dot{e}_2) \rightsquigarrow \texttt{ap}(e_1; e_2) : \tau'} \tag{3.6c}$$

Observe that, in each of these rules, the ce-expression form and the expanded expression form in the conclusion correspond, and the premises correspond to those of the corresponding typing rule, i.e. Rules (3.2a) through (3.2c), respectively. The expression splicing scene, $\mathbb{E}$, passes opaquely through these rules.

We can express this scheme more precisely with the following rule transformation. For each rule in Rules (3.2),

$$\frac{J_1 \quad \cdots \quad J_k}{J}$$

the corresponding candidate expansion expression validation rule is

$$\frac{\mathcal{C}(J_1) \quad \cdots \quad \mathcal{C}(J_k)}{\mathcal{C}(J)}$$

where

$$\begin{aligned}
\mathcal{C}(\Delta \vdash \tau\ \mathsf{type}) &= \Delta \vdash^{\mathsf{ts}(\mathbb{E})} \mathcal{C}(\tau) \rightsquigarrow \tau\ \mathsf{type} \\
\mathcal{C}(\Delta\ \Gamma \vdash e : \tau) &= \Delta\ \Gamma \vdash^{\mathbb{E}} \mathcal{C}(e) \rightsquigarrow e : \tau \\
\mathcal{C}(\{J_i\}_{i \in L}) &= \{\mathcal{C}(J_i)\}_{i \in L}
\end{aligned}$$

and where:

- $\mathcal{C}(\tau)$ is defined as follows:
  - When τ is of definite form, $\mathcal{C}(\tau)$ is defined as in Sec. 3.2.8.
  - When τ is of indefinite form, $\mathcal{C}(\tau)$ is a uniquely corresponding metavariable of sort CETyp also of indefinite form. For example, $\mathcal{C}(\tau_1) = \grave{\tau}_1$ and $\mathcal{C}(\tau_2) = \grave{\tau}_2$.
- $\mathcal{C}(e)$ is defined as follows
  - When e is of definite form, $\mathcal{C}(e)$ is defined as in Sec. 3.2.8.
  - When e is of indefinite form, $\mathcal{C}(e)$ is a uniquely corresponding metavariable of sort CEExp also of indefinite form. For example, $\mathcal{C}(e_1) = \grave{e}_1$ and $\mathcal{C}(e_2) = \grave{e}_2$.

It is instructive to use this rule transformation to generate Rules (3.6a) through (3.6c) above. We omit the remaining rules for shared forms, i.e. Rules (3.6d) through (3.6k).

The following lemma establishes that each well-typed expanded expression, e, can be expressed as a valid ce-expression, $\mathcal{C}(e)$, that is assigned the same type under any expression splicing scene.

Theorem 3.20 (Candidate Expansion Expression Expressibility)**.** *If* $\Delta\ \Gamma \vdash e : \tau$ *then* $\Delta\ \Gamma \vdash^{\mathbb{E}} \mathcal{C}(e) \rightsquigarrow e : \tau$.

Proof. By rule induction over Rules (3.2). The rule transformation above guarantees that this lemma holds by construction. In particular, in each case, we apply Lemma 3.18 to or over each type formation premise, the IH to or over each typing premise, then apply the corresponding ce-expression validation rule in Rules (3.6a) through (3.6k). □

Notice that in Rule (3.6a), only variables tracked by the candidate 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. 3.1.3 – ueTSMs cannot impose "hidden constraints" on the application site unexpanded typing context, because the variable bindings at the application site are not directly available to candidate expansions.

References to Spliced Unexpanded Expressions The only ce-expression form that does not correspond to an expanded expression form is $\texttt{cesplicede}[m;n]$, 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 (3.6l) governs this form:

$$\frac{\mathsf{parseUExp}(\mathsf{subseq}(b;m;n)) = \hat{e} \qquad \langle \mathcal{D};\Delta_{\text{app}}\rangle\ \langle \mathcal{G};\Gamma_{\text{app}}\rangle \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau \\ \Delta \cap \Delta_{\text{app}} = \emptyset \qquad \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma_{\text{app}}) = \emptyset}{\Delta\ \Gamma \vdash^{\langle \mathcal{D};\Delta_{\text{app}}\rangle;\langle \mathcal{G};\Gamma_{\text{app}}\rangle;\hat{\Psi};b} \texttt{cesplicede}[m;n] \rightsquigarrow e : \tau} \tag{3.6l}$$

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 $\mathsf{parseUExp}(b)$, described in Sec. 3.2.4, to produce the referenced spliced unexpanded expression, $\hat{e}$.

The second premise of Rule (3.6l) types and expands the spliced unexpanded expression $\hat{e}$ assuming the application site contexts that appear in the expression splicing scene. The hypotheses in the candidate expansion type formation context, Δ, and typing context, Γ, are not made available to $\hat{e}$.

The third premise of Rule (3.6l) imposes the constraint that the candidate expansion's type formation context, Δ, be disjoint from the application site type formation context, Δ_{app}. Similarly, the fourth premise requires that the candidate 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 ce-expression that the reference to the spliced unexpanded expression appears within.

This achieves *expansion-independent splicing* as described in Sec. 3.1.3 – the TSM provider can choose variable names freely within a candidate expansion, because the language prevents them from shadowing those at the application site (by α-varying the candidate expansion as needed).

3.2.10 Metatheory

For the judgements we have defined to form a sensible language, we must have that typed expansion and candidate expansion expression validation be consistent with typing. Formally, this can be expressed as follows.

Theorem 3.21 (Typed Expansion)**.** *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 (3.4) and Rules (3.6).

The proof of part 1 proceeds by inducting over the typed expansion assumption. 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 (3.4a)**.** We have:

(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 (3.2a)

Case (3.4b)**.** We have:

(1)	$\hat{e} = \texttt{ulam}\{\hat{\tau}_1\}(\hat{x}.\hat{e}')$	by assumption
(2)	$e = \texttt{lam}\{\tau_1\}(x.e')$	by assumption
(3)	$\tau = \texttt{parr}(\tau_1; \tau_2)$	by assumption
(4)	$\hat{\Delta} \vdash \hat{\tau}_1 \rightsquigarrow \tau_1\ \mathsf{type}$	by assumption
(5)	$\hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau_1 \vdash_{\hat{\Psi}} \hat{e}' \rightsquigarrow e' : \tau_2$	by assumption
(6)	$\Delta \vdash \tau_1\ \mathsf{type}$	by Lemma 3.12 on (4)
(7)	$\Delta\ \Gamma, x : \tau_1 \vdash e' : \tau_2$	by IH, part 1 on (5)
(8)	$\Delta\ \Gamma \vdash \texttt{lam}\{\tau_1\}(x.e') : \texttt{parr}(\tau_1; \tau_2)$	by Rule (3.2b) on (6) and (7)

Case (3.4c)**.** We have:

(1)	$\hat{e} = \texttt{uap}(\hat{e}_1; \hat{e}_2)$	by assumption
(2)	$e = \texttt{ap}(e_1; e_2)$	by assumption
(3)	$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e}_1 \rightsquigarrow e_1 : \texttt{parr}(\tau_1; \tau)$	by assumption
(4)	$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e}_2 \rightsquigarrow e_2 : \tau_1$	by assumption
(5)	$\Delta\ \Gamma \vdash e_1 : \texttt{parr}(\tau_1; \tau)$	by IH, part 1 on (3)
(6)	$\Delta\ \Gamma \vdash e_2 : \tau_1$	by IH on (4)
(7)	$\Delta\ \Gamma \vdash \texttt{ap}(e_1; e_2) : \tau$	by Rule (3.2c) on (5) and (6)

Case (3.4d) through (3.4k)**.** These cases follow analagously, i.e. we apply Lemma 3.12 to or over the type expansion premises and the IH, part 1, to or over the typed expression expansion premises and then apply the corresponding typing rule in Rules (3.2d) through (3.2k).

Case (3.4l)**.** We have

(1)	$\hat{e} = \texttt{usyntaxue}\{e_{\text{parse}}\}\{\hat{\tau}'\}(\hat{a}.\hat{e}')$	by assumption

(2) $\hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'\ \mathsf{type}$ — by assumption
(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{uetsm}(\tau'; e_{\mathrm{parse}})} \hat{e}' \rightsquigarrow e : \tau$ — by assumption
(4) $\Delta\ \Gamma \vdash e : \tau$ — by IH, part 1 on (3)

Case (3.4m)**.** We have

(1) $\hat{e} = \mathtt{uapuetsm}[b]\,[\hat{a}]$ — by assumption
(2) $\mathcal{A} = \mathcal{A}', \hat{a} \rightsquigarrow a$ — by assumption
(3) $\Psi = \Psi', a \hookrightarrow \mathtt{uetsm}(\tau; e_{\mathrm{parse}})$ — by assumption
(4) $b \downarrow e_{\mathrm{body}}$ — by assumption
(5) $e_{\mathrm{parse}}(e_{\mathrm{body}}) \Downarrow \mathtt{inj}\ \mathtt{Success} \cdot e_{\mathrm{cand}}$ — by assumption
(6) $e_{\mathrm{cand}} \uparrow_{\mathsf{CEExp}} \grave{e}$ — 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 identity
(9) $\emptyset \cap \mathrm{dom}(\Gamma) = \emptyset$ — by finite set intersection identity
(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 definition of finite set and finite function union over (10)

The second part of the theorem proceeds by induction over the candidate expansion expression validation assumption as follows. In the following cases, let $\hat{\Delta}_{\mathrm{app}} = \langle \mathcal{D}; \Delta_{\mathrm{app}} \rangle$ and $\hat{\Gamma}_{\mathrm{app}} = \langle \mathcal{G}; \Gamma_{\mathrm{app}} \rangle$ and $\hat{\Psi} = \langle \mathcal{A}; \Psi \rangle$.

Case (3.6a)**.** We have

(1) $\grave{e} = x$ — by assumption
(2) $e = x$ — by assumption
(3) $\Gamma = \Gamma', x : \tau$ — by assumption
(4) $\Delta \cup \Delta_{\mathrm{app}}\ \Gamma', x : \tau \vdash x : \tau$ — by Rule (3.2a)
(5) $\Delta \cup \Delta_{\mathrm{app}}\ \Gamma', x : \tau \cup \Gamma_{\mathrm{app}} \vdash x : \tau$ — by Lemma 3.2 over Γ_{app} to (4)

Case (3.6b)**.** We have

(1) $\grave{e} = \mathtt{celam}\{\grave{\tau}_1\}(x.\grave{e}')$ — by assumption
(2) $e = \mathtt{lam}\{\tau_1\}(x.e')$ — by assumption
(3) $\tau = \mathtt{parr}(\tau_1; \tau_2)$ — by assumption
(4) $\Delta \vdash^{\hat{\Delta}_{\mathrm{app}}; b} \grave{\tau}_1 \rightsquigarrow \tau_1\ \mathsf{type}$ — by assumption
(5) $\Delta\ \Gamma, x : \tau_1 \vdash^{\hat{\Delta}_{\mathrm{app}}; \hat{\Gamma}_{\mathrm{app}}; \hat{\Psi}; b} \grave{e}' \rightsquigarrow e' : \tau_2$ — by assumption
(6) $\Delta \cap \Delta_{\mathrm{app}} = \emptyset$ — by assumption

(7) $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma_{\mathrm{app}}) = \emptyset$ by assumption
(8) $x \notin \mathrm{dom}(\Gamma_{\mathrm{app}})$ by identification convention
(9) $\mathrm{dom}(\Gamma, x : \tau_1) \cap \mathrm{dom}(\Gamma_{\mathrm{app}}) = \emptyset$ by (7) and (8)
(10) $\Delta \cup \Delta_{\mathrm{app}} \vdash \tau_1$ type by Lemma 3.19 on (4)
(11) $\Delta \cup \Delta_{\mathrm{app}}\ \Gamma, x : \tau_1 \cup \Gamma_{\mathrm{app}} \vdash e' : \tau_2$ by IH, part 2 on (5), (6) and (9)
(12) $\Delta \cup \Delta_{\mathrm{app}}\ \Gamma \cup \Gamma_{\mathrm{app}}, x : \tau_1 \vdash e' : \tau_2$ by exchange over Γ_{app} on (11)
(13) $\Delta \cup \Delta_{\mathrm{app}}\ \Gamma \cup \Gamma_{\mathrm{app}} \vdash \mathtt{lam\{}\tau_1\mathtt{\}(}x.e'\mathtt{)} : \mathtt{parr(}\tau_1;\tau_2\mathtt{)}$ by Rule (3.2b) on (10) and (12)

Case (3.6c)**.** We have

(1) $\grave{e} = \mathtt{ceap(}\grave{e}_1;\grave{e}_2\mathtt{)}$ by assumption
(2) $e = \mathtt{ap(}e_1;e_2\mathtt{)}$ by assumption
(3) $\Delta\ \Gamma \vdash^{\hat{\Delta}_{\mathrm{app}};\hat{\Gamma}_{\mathrm{app}};\hat{\Psi};b} \grave{e}_1 \rightsquigarrow e_1 : \mathtt{parr(}\tau_2;\tau\mathtt{)}$ by assumption
(4) $\Delta\ \Gamma \vdash^{\hat{\Delta}_{\mathrm{app}};\hat{\Gamma}_{\mathrm{app}};\hat{\Psi};b} \grave{e}_2 \rightsquigarrow e_2 : \tau_2$ by assumption
(5) $\Delta \cap \Delta_{\mathrm{app}} = \emptyset$ by assumption
(6) $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma_{\mathrm{app}}) = \emptyset$ by assumption
(7) $\Delta \cup \Delta_{\mathrm{app}}\ \Gamma \cup \Gamma_{\mathrm{app}} \vdash e_1 : \mathtt{parr(}\tau_2;\tau\mathtt{)}$ by IH, part 2 on (3), (5) and (6)
(8) $\Delta \cup \Delta_{\mathrm{app}}\ \Gamma \cup \Gamma_{\mathrm{app}} \vdash e_2 : \tau_2$ by IH, part 2 on (4), (5) and (6)
(9) $\Delta \cup \Delta_{\mathrm{app}}\ \Gamma \cup \Gamma_{\mathrm{app}} \vdash \mathtt{ap(}e_1;e_2\mathtt{)} : \tau$ by Rule (3.2c) on (7) and (8)

Case (3.6d)**.** We have

(1) $\grave{e} = \mathtt{cetlam(}t.\grave{e}'\mathtt{)}$ by assumption
(2) $e = \mathtt{tlam(}t.e'\mathtt{)}$ by assumption
(3) $\tau = \mathtt{all(}t.\tau'\mathtt{)}$ by assumption
(4) $\Delta, t\ \mathsf{type}\ \Gamma \vdash^{\hat{\Delta}_{\mathrm{app}};\hat{\Gamma}_{\mathrm{app}};\hat{\Psi};b} \grave{e}' \rightsquigarrow e' : \tau'$ by assumption
(5) $\Delta \cap \Delta_{\mathrm{app}} = \emptyset$ by assumption
(6) $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma_{\mathrm{app}}) = \emptyset$ by assumption
(7) $t\ \mathsf{type} \notin \Delta_{\mathrm{app}}$ by identification convention
(8) $\Delta, t\ \mathsf{type} \cap \Delta_{\mathrm{app}} = \emptyset$ by (5) and (7)
(9) $\Delta, t\ \mathsf{type} \cup \Delta_{\mathrm{app}}\ \Gamma \cup \Gamma_{\mathrm{app}} \vdash e' : \tau'$ by IH, part 2 on (4), (8) and (6)
(10) $\Delta \cup \Delta_{\mathrm{app}}, t\ \mathsf{type}\ \Gamma \cup \Gamma_{\mathrm{app}} \vdash e' : \tau'$ by exchange over Δ_{app} on (9)
(11) $\Delta \cup \Delta_{\mathrm{app}}\ \Gamma \cup \Gamma_{\mathrm{app}} \vdash \mathtt{tlam(}t.e'\mathtt{)} : \mathtt{all(}t.\tau'\mathtt{)}$ by Rule (3.2d) on (10)

Case (3.6e) through (3.6k)**.** These cases follow analagously, i.e. we apply the IH, part 2 to all ce-expression validation judgements, Lemma 3.19 to all ce-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 (3.2e) through (3.2k).

Case (3.6l)**.** We have

(1) $\grave{e} = \texttt{cesplicede}[m;n]$	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 on (3)
(7) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash e : \tau$	by Lemma 3.2 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 $\hat{e}$,

$$\begin{aligned}
\|\hat{x}\| &= 0 \\
\|\texttt{ulam\{}\hat{\tau}\texttt{\}(}\hat{x}.\hat{e}\texttt{)}\| &= \|\hat{e}\| \\
\|\texttt{uap(}\hat{e}_1;\hat{e}_2\texttt{)}\| &= \|\hat{e}_1\| + \|\hat{e}_2\| \\
\|\texttt{utlam(}\hat{t}.\hat{e}\texttt{)}\| &= \|\hat{e}\| \\
\|\texttt{utap\{}\hat{\tau}\texttt{\}(}\hat{e}\texttt{)}\| &= \|\hat{e}\| \\
\|\texttt{ufold\{}\hat{t}.\hat{\tau}\texttt{\}(}\hat{e}\texttt{)}\| &= \|\hat{e}\| \\
\|\texttt{uunfold(}\hat{e}\texttt{)}\| &= \|\hat{e}\| \\
\|\texttt{utpl}[L]\texttt{(}\{i \hookrightarrow \hat{e}_i\}_{i\in L}\texttt{)}\| &= \sum_{i\in L} \|\hat{e}_i\| \\
\|\texttt{upr[}\ell\texttt{](}\hat{e}\texttt{)}\| &= \|\hat{e}\| \\
\|\texttt{uin}[L]\texttt{[}\ell\texttt{]\{\{}i \hookrightarrow \hat{\tau}_i\}_{i\in L}\texttt{\}(}\hat{e}\texttt{)}\| &= \|\hat{e}\| \\
\|\texttt{ucase}[L]\texttt{\{}\hat{\tau}\texttt{\}(}\hat{e};\{i \hookrightarrow \hat{x}_i.\hat{e}_i\}_{i\in L}\texttt{)}\| &= \|\hat{e}\| + \sum_{i\in L} \|\hat{e}_i\| \\
\|\texttt{usyntaxue\{}e_{\text{parse}}\texttt{\}\{}\hat{\tau}\texttt{\}(}\hat{a}.\hat{e}\texttt{)}\| &= \|\hat{e}\| \\
\|\texttt{uapuetsm}[b]\texttt{[}\hat{a}\texttt{]}\| &= \|b\|
\end{aligned}$$

The only case in the proof of part 1 that invokes part 2 is Case (3.4m). There, we have that the metric remains stable:

$$\begin{aligned}&\|\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uetsm}(\tau;e_{\text{parse}})} \texttt{uapuetsm}[b]\,\texttt{[}\hat{a}\texttt{]} \rightsquigarrow e : \tau\|\\ =&\|\varnothing\ \varnothing \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uetsm}(\tau;e_{\text{parse}});b} \grave{e} \rightsquigarrow e : \tau\|\\ =&\|b\|\end{aligned}$$

The only case in the proof of part 2 that invokes part 1 is Case (3.6l). 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:

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

i.e. by the definitions above,

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

This is established by appeal to the following two conditions. The first condition simply states that subsequences of b are no longer than b.

Condition 3.22 (Body Subsequencing)**.** *If* $\mathsf{subseq}(b;m;n) = b'$ *then* $\|b'\| \leq \|b\|$.

The second 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 3.23 (Expression Parsing Monotonicity)**.** *If* $\mathsf{parseUExp}(b) = \hat{e}$ *then* $\|\hat{e}\| < \|b\|$.

Combining Conditions 3.22 and 3.23, we have that $\|\hat{e}\| < \|b\|$ as needed. □

Chapter 4

Unparameterized Pattern TSMs (upTSMs)

In Chapter 3, we considered 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. In full-scale functional languages like ML and Haskell, values are deconstructed by *pattern matching* over their structure. 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:

```
1 fun read_example_rx(r : Rx) : (string * Rx) option =>
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. When the match expression is evaluated, the value of the scrutinee is matched against each pattern sequentially. If the value matches, evaluation takes the corresponding branch. Variables in patterns match any value of the appropriate type. In the corresponding branch, the variable stands for that value. Variables can each appear only once in a pattern. For example, on Line 3, the pattern `Seq(Str(name), Seq(Str ": ", ssn))` matches values of the form `Seq(Str(`e_1`), Seq(Str ": ",` e_2`))`, where e_1 is a value of type `string` and e_2 is a value of type `Rx`. The variables `name` and `ssn` stand for the values of e_1 and e_2, respectively, in `Some (name, ssn)`. On Line 4, 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 are not matched by any pattern in the rule sequence). In the example above, our use of the wildcard pattern ensures that match failure cannot occur. 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.

It is also possible to statically decide when a rule is *redundant* relative to the preceding rules, i.e. when there does not exist a value matched by that rule but not matched by any of 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. $\text{miniVerse}_{\text{UE}}$ from the previous section) and decreases syntactic cost in situations where eliminators would need to be nested. There remains room for improvement, however, because complex patterns sometimes individually have high syntactic cost. In Sec. 2.3.2, we considered a hypothetical dialect of ML called ML+Rx that built in derived syntax both for constructing and pattern matching over values of the recursive labeled sum type `Rx`. In ML+Rx, we can express the example above at lower syntactic cost as follows:

```
1 fun read_example_rx(r : Rx) : (string * Rx) option =>
2   match r with
3     /@name: %ssn/ => Some (name, ssn)
4   | _ => None
```

Dialect formation is not a modular approach, for the reasons discussed in Chapter 1, so we seek language constructs that allow us to decrease the syntactic cost of expressing complex patterns to a similar degree.

Expression TSMs – introduced in Chapter 3 – can decrease the syntactic cost of constructing a value of a specified type. 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 **unparameterized pattern TSMs** (upTSMs), 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).

4.1 Pattern TSMs By Example

The organization of the remainder of this chapter mirrors that of Chapter 3. We begin in this section with a "tutorial-style" introduction to upTSMs in VerseML. In particular, we discuss an upTSM for patterns matching values of type `Rx`. In the next section, we specify a reduced formal system based on $\text{miniVerse}_{\text{UE}}$ called $\text{miniVerse}_{\text{U}}$ that makes the intuitions developed here mathematically precise.

[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.

4.1.1 Usage

The VerseML function `read_example_rx` defined at the beginning of this chapter can be concretely expressed at lower syntactic cost by applying a upTSM, `$rx`, as follows:

```
1 fun read_example_rx(r : Rx) : (string * Rx) option =>
2   match r with
3     $rx /@name: %ssn/ => Some (name, ssn)
4   | _ => None
```

Like expression TSMs, pattern TSMs are applied to *generalized literal forms* (see Figure 3.1). Generalized literal forms are left unparsed when patterns are first parsed. During the subsequent *typed expansion* process, the pattern TSM parses the body of the literal form to generate a *candidate expansion*. The language validates the candidate 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 can be performed post-expansion in the usual way, so we do not need to consider them further here.

4.1.2 Definition

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

```
syntax $rx at Rx for patterns {
  static fn(body : Body) : CEPat ParseResult =>
    (* regex pattern parser here *)
}
```

This definition first names the pattern TSM. Pattern TSM names, like expression TSM names, must begin with the dollar symbol (`$`) to distinguish them from labels. Pattern TSM names and expression TSM names are tracked separately, i.e. an expression TSM and a pattern TSM can have the same name without conflict (as is the case here – the expression TSM described in Sec. 3.1.2 is also named `$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). Because defining both an expression TSM and a pattern TSM with the same name at the same type is a common idiom, VerseML provides a primitive derived form for combining their definitions:

```
syntax $rx at Rx for expressions {
  static fn(body : Body) : CEExp ParseResult =>
    (* regex expression parser here *)
} for patterns {
  static fn(body : Body) : CEPat ParseResult =>
    (* regex pattern parser here *)
```

```
type CEPat = Wild
           | (* ... *)
           | Spliced of IndexRange
```

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

```
}
```

Pattern TSMs, like expression TSMs, must specify a static *parse function*, delimited by curly braces in the concrete syntax. For a pattern TSM, the parse function must be of type `Body -> CEPat ParseResult`. The input type, `Body`, gives the parse function access to the body of the provided literal form, and is defined as in Sec. 3.1.2 as a synonym for the type `string`. The output type, `CEPat ParseResult`, is the parameterized type constructor `ParseResult`, defined in Figure 3.2, applied to the type `CEPat` defined in Figure 4.1. So if parsing succeeds, the pattern TSM returns a value of the form `Success` e_{cand} where e_{cand} is a value of type `CEPat` that we call the *encoding of the candidate expansion*. If parsing fails, then the pattern TSM returns a value constructed by `ParseError` and equipped with an error message and error location.

The type `CEPat` is analagous to the types `CEExp` and `CETyp` defined in Figure 3.3. It encodes the abstract syntax of VerseML patterns (in Figure 4.1, some constructors are elided for concision), with the exception of variable patterns (for reasons explained in Sec. 4.1.5 below), and includes an additional constructor, `Spliced`, for referring to spliced subpatterns by their position within the parse stream, discussed next.

4.1.3 Splicing

Patterns that appear directly within the literal body of an unexpanded pattern are called *spliced subpatterns*. For example, the patterns `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 treated as a spliced subpatern (here, by recognizing the characters `@` or `%` followed by a variable or parenthesized pattern), it can refer to it within the candidate expansion that it construct a reference to it for use within the candidate expansion it generates using the `Spliced` constructor of the `CEPat` type shown in Figure 4.1. The `Spliced` constructor requires a value of type `IndexRange` because spliced subpatterns are referred to indirectly by their position within the literal body. This prevents pattern TSMs from "forging" a spliced subpattern (i.e. claiming that some pattern is a spliced subpattern, even though it does not appear in the literal body).

The candidate expansion generated by the pattern TSM `$rx` for the example above, if written in a hypothetical concrete syntax where references to spliced subpatterns 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.

4.1.4 Typing

The language validates candidate expansion before a final expansion is generated. One aspect of candidate expansion validation is checking the candidate expansion against the type annotation specified by the pattern TSM, e.g. the type `Rx` in the example above.

4.1.5 Hygiene

In order to check that the candidate expansion is well-typed, the language must parse, type and expand the spliced subpatterns that the candidate expansion refers to (by their position within the literal body, cf. above). To maintain a useful binding discipline, i.e. to allow programmers to reason about variable binding without examining expansions directly, the validation process allows variables (e.g. `name` and `ssn` above) to occur only in spliced subpatterns (just as variables bound at the use site can only appear in spliced subexpressions when using TSMs). Indeed, there is no constructor for the type `CEPat` corresponding to a variable pattern. This protection against "hidden bindings" is beneficial because it leaves variable naming entirely up to the client of the pattern TSM. A pattern TSM cannot inadvertently shadow a binding at the application site.

4.1.6 Final Expansion

If validation succeeds, the semantics generates the *final expansion* of the pattern from the candidate expansion by replacing the references to spliced subpatterns with their final expansions. For example, the final expansion of `$rx /@name: %ssn/` is:

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

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

To make the intuitions developed in the previous section about pattern TSMs precise, we now introduce miniVerse$_{\mathbf{U}}$, a reduced language with support for both ueTSMs and upTSMs. Like miniVerse$_{\mathbf{UE}}$, miniVerse$_{\mathbf{U}}$ consists of an *inner core* and an *outer surface*.

4.2.1 Syntax of the Inner Core

The *inner core* of miniVerse$_{\mathbf{U}}$ consists of *types*, τ, *expanded expressions*, e, *expanded rules*, r, and *expanded patterns*, p. Their syntax is specified by the syntax chart in Figure 4.2. The inner core of miniVerse$_{\mathbf{U}}$ forms a pure language and differs from the inner core of miniVerse$_{\mathbf{UE}}$ only in that the case analysis operator has been replaced by the pattern matching operator[2], so we will gloss some definitions that would be expressed identically to those in Sec. 3.2. The new constructs are highlighted in gray in Figure 4.2. Our

[2]We retain the projection operator because it has lower syntactic cost than pattern matching when only a single field from a labeled tuple is needed.

formulation of the semantics of pattern matching is based on Harper's formulation in *Practical Foundations for Programming Languages, First Edition* [37].[3]

4.2.2 Statics of the Inner Core

The *statics of the inner core* is specified by judgements of the following form:

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

The types of $\mathsf{miniVerse}_{\mathbf{U}}$ are exactly those of $\mathsf{miniVerse}_{\mathbf{UE}}$, described in Sec. 3.2, so the *type formation judgement*, $\Delta \vdash \tau$ type, is inductively defined by Rules (3.1).

The *typing judgement*, $\Delta\ \Gamma \vdash e : \tau$, assigns types to expressions and is inductively defined by Rules (4.1), which consist of:

- Rules written identically to Rules (3.2a) through (3.2j). We will refer to these rules as Rules (4.1a) through (4.1j).
- The following rule for match expressions:

$$\frac{\Delta\ \Gamma \vdash e : \tau \qquad \Delta \vdash \tau' \text{ type} \qquad \{\Delta\ \Gamma \vdash r_i : \tau \Mapsto \tau'\}_{1 \leq i \leq n}}{\Delta\ \Gamma \vdash \texttt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \leq i \leq n}) : \tau'} \tag{4.1k}$$

The first premise of Rule (4.1k) 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.[4] 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'. This is expressed by the *rule typing judgement*, $\Delta\ \Gamma \vdash r : \tau \Mapsto \tau'$, which is defined mutually with Rules (4.1) by the following rule:

$$\frac{\Delta \vdash p : \tau \dashv\mid \Upsilon \qquad \Delta\ \Gamma \cup \Upsilon \vdash e : \tau'}{\Delta\ \Gamma \vdash \texttt{rule}(p.e) : \tau \Mapsto \tau'} \tag{4.2}$$

The premises of Rule (4.2) can be understood as follows, in order:

1. The first premise invokes the *pattern typing judgement*, $\Delta \vdash p : \tau \dashv\mid \Upsilon$, 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*, Υ. We use the metavariable Υ (i.e. "upsilon") rather than Γ only to emphasize the distinct role of the typing context in the pattern typing judgement – algorithmically, it is the "output" of the judgement.

[3] The chapter on pattern matching has, of this writing, been removed from the draft second edition of *PFPL*, but a copy of the first edition can be found online.

[4] The second premise of Rule (4.1k), 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.

Sort			Operational Form	Stylized Form	Description
Typ	τ	::=	t	t	variable
			$\texttt{parr}(\tau;\tau)$	$\tau \rightharpoonup \tau$	partial function
			$\texttt{all}(t.\tau)$	$\forall t.\tau$	polymorphic
			$\texttt{rec}(t.\tau)$	$\mu t.\tau$	recursive
			$\texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})$	$\langle \{i \hookrightarrow \tau_i\}_{i\in L} \rangle$	labeled product
			$\texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i\in L})$	$[\{i \hookrightarrow \tau_i\}_{i\in L}]$	labeled sum
Exp	e	::=	x	x	variable
			$\texttt{lam}\{\tau\}(x.e)$	$\lambda x{:}\tau.e$	abstraction
			$\texttt{ap}(e;e)$	$e(e)$	application
			$\texttt{tlam}(t.e)$	$\Lambda t.e$	type abstraction
			$\texttt{tap}\{\tau\}(e)$	$e[\tau]$	type application
			$\texttt{fold}\{t.\tau\}(e)$	$\texttt{fold}(e)$	fold
			$\texttt{unfold}(e)$	$\texttt{unfold}(e)$	unfold
			$\texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L})$	$\langle \{i \hookrightarrow e_i\}_{i\in L} \rangle$	labeled tuple
			$\texttt{pr}[\ell](e)$	$e \cdot \ell$	projection
			$\texttt{in}[L][\ell]\{\{i \hookrightarrow \tau_i\}_{i\in L}\}(e)$	$\texttt{inj}\ \ell \cdot e$	injection
			$\texttt{match}[n]\{\tau\}(e;\{r_i\}_{1\le i\le n})$	$\texttt{match}\ e\ \{r_i\}_{1\le i\le n}$	match
Rule	r	::=	$\texttt{rule}(p.e)$	$p \Rightarrow e$	rule
Pat	p	::=	x	x	variable pattern
			$\texttt{wildp}$	$\_$	wildcard pattern
			$\texttt{foldp}(p)$	$\texttt{fold}(p)$	fold pattern
			$\texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i\in L})$	$\langle \{i \hookrightarrow p_i\}_{i\in L} \rangle$	labeled tuple pattern
			$\texttt{inp}[\ell](p)$	$\texttt{inj}\ \ell \cdot p$	injection pattern

Figure 4.2: Syntax of types and expanded expressions, rules and patterns (collectively, expanded terms) in miniVerse$_{\mathbf{U}}$. We adopt the metatheoretic conventions established for our specification of miniVerse$_{\mathbf{UE}}$ in Sec. 3.2 without restating them. We write $\{r_i\}_{1\le i\le n}$ for sequences of $n \geq 0$ rule arguments and $p.e$ for expressions binding the variables that appear in the pattern p. Types and expanded terms are identified up to α-equivalence.

The pattern typing judgement is inductively defined by the following rules:

$$\frac{}{\Delta \vdash x : \tau \dashv\Vdash x : \tau} \tag{4.3a}$$

$$\frac{}{\Delta \vdash \texttt{wildp} : \tau \dashv\Vdash \emptyset} \tag{4.3b}$$

$$\frac{\Delta \vdash p : [\texttt{rec}(t.\tau)/t]\tau \dashv\Vdash \mathcal{Y}}{\Delta \vdash \texttt{foldp}(p) : \texttt{rec}(t.\tau) \dashv\Vdash \mathcal{Y}} \tag{4.3c}$$

$$\frac{\{\Delta \vdash p_i : \tau_i \dashv\Vdash \mathcal{Y}_i\}_{i \in L}}{\Delta \vdash \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\Vdash \cup_{i \in L} \mathcal{Y}_i} \tag{4.3d}$$

$$\frac{\Delta \vdash p : \tau \dashv\Vdash \mathcal{Y}}{\Delta \vdash \texttt{inp}[\ell](p) : \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\Vdash \mathcal{Y}} \tag{4.3e}$$

Rule (4.3a) specifies that a variable pattern, x, matches values of any type, τ, and generates the hypothesis that x has the type τ.

Rule (4.3b) specifies that a wildcard pattern also matches values of any type, τ, but wildcard patterns generate no hypotheses.

Rule (4.3c) specifies that a fold pattern, $\texttt{foldp}(p)$, matches values of the recursive type $\texttt{rec}(t.\tau)$ and generates hypotheses $\mathcal{Y}$ if p matches values of a single unrolling of the recursive type, $[\texttt{rec}(t.\tau)/t]\tau$, and generates hypotheses $\mathcal{Y}$.

Rule (4.3d) 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 $\mathcal{Y}_i$. The conclusion of the rule gathers these hypotheses into a single pattern typing context, $\cup_{i \in L} \mathcal{Y}_i$. The definition of typing context extension, applied iteratively here, implicitly requires that the pattern typing contexts $\mathcal{Y}_i$ be mutually disjoint, i.e.

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

Rule (4.3e) specifies that an injection pattern, $\texttt{inp}[\ell](p)$, 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, generating hypotheses $\mathcal{Y}$ if p matches value of type τ and generates hypotheses $\mathcal{Y}$.

2. The final premise of Rule (4.2) extends the typing context, Γ, with the hypotheses generated by pattern typing, $\mathcal{Y}$, and checks the branch expression, e, against the branch type, τ'.

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 4.1 (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* $\Delta \vdash p : \tau \dashv\mid \mathcal{Y}$ *then* Δ, t type $\vdash p : \tau \dashv\mid \mathcal{Y}$.

Proof Sketch.

1. By rule induction over Rules (3.1).
2. By mutual rule induction over Rules (4.1) and Rule (4.2) and part 1.
3. By mutual rule induction over Rules (4.1) and Rule (4.2) and part 1.
4. By rule induction over Rules (4.3).

□

The pattern typing judgement is a *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.

We assume that renaming of bound identifiers, α-equivalence and substitution can be defined essentially as in *PFPL* [38], modified only so that binders involving patterns bind exactly those variables mentioned in the pattern in some arbitrary deterministic order. 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 4.2 (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 (3.1).
2. By mutual rule induction over Rules (4.1) and Rule (4.2).
3. By mutual rule induction over Rules (4.1) and Rule (4.2).

□

The Decomposition Lemma is the converse of the Substitution Lemma.

Lemma 4.3 (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 $\Delta\ \Gamma \vdash [e'/x]r : \tau \Rightarrow \tau''$ *and* $\Delta\ \Gamma \vdash e' : \tau'$ *then* $\Delta\ \Gamma, x : \tau' \vdash r : \tau \Rightarrow \tau''$.

Proof Sketch.

1. By rule induction over Rules (3.1) 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 (4.1) and Rule (4.2) 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 $\Delta\ [\tau'/t]\Gamma \vdash [\tau'/t]r : [\tau'/t]\tau \Rightarrow [\tau'/t]\tau''$ does not depend on the form of τ'.
3. By mutual rule induction over Rules (4.1) and Rule (4.2) and case analysis over the definition of substitution. In all cases, the derivation of $\Delta\ \Gamma \vdash [e'/x]e : \tau$ or $\Delta\ \Gamma \vdash [e'/x]r : \tau \Rightarrow \tau''$ 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 4.4 (Pattern Regularity)**.** *If* $\Delta \vdash p : \tau \dashv\Vdash \mathcal{Y}$ *and* $\Delta \vdash \tau$ type *then* $\Delta \vdash \mathcal{Y}$ ctx.

Proof. By rule induction over Rules (4.3).

Case (4.3a)**.** We have:

(1) $p = x$	by assumption
(2) $\mathcal{Y} = x : \tau$	by assumption
(3) $\Delta \vdash \tau$ type	by assumption
(4) $\Delta \vdash x : \tau$ ctx	by Definition 3.1 on (3)

Case (4.3b)**.** We have:

(1) $\mathcal{Y} = \emptyset$	by assumption
(2) $\Delta \vdash \emptyset$ ctx	by Definition 3.1

Case (4.3d)**.** We have:

(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) $\mathcal{Y} = \cup_{i \in L} \mathcal{Y}_i$	by assumption
(4) $\{\Delta \vdash p_i : \tau_i \dashv\Vdash \mathcal{Y}_i\}_{i \in L}$	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 (3.1e) on (5)
(7) $\{\Delta \vdash \mathcal{Y}_i$ ctx$\}_{i \in L}$	by IH over (4) and (6)

(8) $\Delta \vdash \cup_{i \in L} \mathcal{Y}_i$ ctx — by Definition 3.1 on (7), then Definition 3.1 again, using the definition of typing context union iteratively

Case (4.3e)**.** We have:

(1) $p = \texttt{inp}[\ell](p')$ — by assumption
(2) $\tau = \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau')$ — by assumption
(3) $\Delta \vdash \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau')$ type — by assumption
(4) $\Delta \vdash p' : \tau' \dashv\mid \mathcal{Y}$ — by assumption
(5) $\Delta \vdash \tau'$ type — by Inversion of Rule (3.1f) on (3)
(6) $\Delta \vdash \mathcal{Y}$ 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 4.5 (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* $\Delta\ \Gamma \vdash r : \tau \Rightarrow \tau'$ *and* $\Delta \vdash \Gamma$ ctx *then* $\Delta \vdash \tau'$ type.

Proof Sketch. By mutual rule induction over Rules (4.1) and Rule (4.2), and Lemma 4.2 and Lemma 4.4. □

4.2.3 Structural Dynamics

The *structural dynamics of* $\mathsf{miniVerse}_{\mathsf{U}}$ 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 4.6 (Iterated Transition)**.** *Iterated transition,* $e \mapsto^* e'$*, is the reflexive, transitive closure of the transition judgement,* $e \mapsto e'$*.*

Definition 4.7 (Evaluation)**.** $e \Downarrow e'$ *iff* $e \mapsto^* e'$ *and* e' val*.*

As in Sec. 3.2.3, our subsequent developments do not make mention of particular rules in the dynamics, nor do they make mention of judgements 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 characterizes well-typed values. Satisfying this condition requires an *eager* (i.e. *by-value*) formulation of the dynamics. This condition is identical to Condition 3.8.

Condition 4.8 (Canonical Forms)**.** *If* $\vdash e : \tau$ *and* e val *then:*

1. *If* $\tau = \texttt{parr}(\tau_1; \tau_2)$ *then* $e = \texttt{lam}\{\tau_1\}(x.e')$ *and* $x : \tau_1 \vdash e' : \tau_2$.
2. *If* $\tau = \texttt{all}(t.\tau')$ *then* $e = \texttt{tlam}(t.e')$ *and* t type $\vdash e' : \tau'$.
3. *If* $\tau = \texttt{rec}(t.\tau')$ *then* $e = \texttt{fold}\{t.\tau'\}(e')$ *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_ℓ*, we have that* $L = L', \ell$ *and* $\tau = \texttt{sum}[L', \ell](\{i \hookrightarrow \tau_i\}_{i \in L'}; \ell \hookrightarrow \tau_\ell)$ *and* $e = \texttt{in}[L', \ell][\ell]\{\{i \hookrightarrow \tau_i\}_{i \in L'}; \ell \hookrightarrow \tau_\ell\}(e')$ *and* $\vdash e' : \tau_\ell$ *and* e' val.

The Preservation condition ensures that evaluation preserves typing.

Condition 4.9 (Preservation)**.** *If* $\vdash e : \tau$ *and* $e \mapsto e'$ *then* $\vdash e' : \tau$.

The Progress condition ensures that evaluation of a well-typed expanded expression cannot "get stuck".

Condition 4.10 (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 exhaustiveness and redundancy properties here, because these can be checked post-expansion and so are also not relevant to our subsequent developments (but see [37] for a formal account).

4.2.4 Syntax of the Outer Surface

A miniVerse$_\mathbf{U}$ 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 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 parseUTyp(b), parseURule(b) and parseUPat(b). The abstract syntax of unexpanded types, expressions, rules and patterns, which form the *outer surface* of miniVerse$_\mathbf{U}$, is defined in Figure 4.3. The full definition of the textual syntax of miniVerse$_\mathbf{U}$ 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 4.11 (Textual Representability)**.** *All of the following must hold:*

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

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
			$\texttt{ulam}\{\hat{\tau}\}(\hat{x}.\hat{e})$	$\lambda \hat{x}{:}\hat{\tau}.\hat{e}$	abstraction
			$\texttt{uap}(\hat{e};\hat{e})$	$\hat{e}(\hat{e})$	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
			$\texttt{ufold}\{\hat{t}.\hat{\tau}\}(\hat{e})$	$\texttt{fold}(\hat{e})$	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
			$\texttt{upr}[\ell](\hat{e})$	$\hat{e} \cdot \ell$	projection
			$\texttt{uin}[L][\ell]\{\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}\}(\hat{e})$	$\texttt{inj}\ \ell \cdot \hat{e}$	injection
			$\texttt{umatch}[n]\{\hat{\tau}\}(\hat{e};\{\hat{r}_i\}_{1 \le i \le n})$	$\texttt{match}\ \hat{e}\ \{\hat{r}_i\}_{1 \le i \le n}$	match
			$\texttt{usyntaxue}\{e\}\{\hat{\tau}\}(\hat{a}.\hat{e})$	$\texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for}\ \texttt{expressions}\ \{e\}\ \texttt{in}\ \hat{e}$	ueTSM definition
			$\texttt{uapuetsm}[b][\hat{a}]$	$\hat{a}\ \texttt{/}b\texttt{/}$	ueTSM application
			$\texttt{usyntaxup}\{e\}\{\hat{\tau}\}(\hat{a}.\hat{e})$	$\texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for}\ \texttt{patterns}\ \{e\}\ \texttt{in}\ \hat{e}$	upTSM definition
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
			$\texttt{uinp}[\ell](\hat{p})$	$\texttt{inj}\ \ell \cdot \hat{p}$	injection pattern
			$\texttt{uapuptsm}[b][\hat{a}]$	$\hat{a}\ \texttt{/}b\texttt{/}$	upTSM application

Figure 4.3: Abstract syntax of unexpanded types, expressions, rules and patterns in $\mathsf{miniVerse}_{\mathsf{U}}$. Metavariable $\hat{t}$ ranges over type sigils, $\hat{x}$ ranges over expression sigils, $\hat{a}$ over TSM names and b over textual sequences, which, when they appear in an unexpanded term, are called literal bodies. Literal bodies might contain spliced subterms that are only "surfaced" during typed expansion, so renaming of bound identifiers and substitution are not defined over unexpanded types and terms.

As in miniVerse$_{\mathbf{UE}}$, 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 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. We refer to these as the *shared forms*. In particular:

- Each type variable, t, maps onto a unique type sigil, written $\widehat{t}$ (pronounced "sigil of t"). Notice the distinction between $\hat{t}$, which is a metavariable ranging over type sigils, and $\widehat{t}$, which is a metafunction, written in stylized form, applied to a type variable to produce a type sigil.
- Each type form, τ, maps onto an unexpanded type form, $\mathcal{U}(\tau)$, according to the definition of $\mathcal{U}(\tau)$ in Sec. 3.2.4.
- Each expression variable, x, maps onto a unique expression sigil, written $\widehat{x}$. Again, notice the distinction between $\hat{x}$ and $\widehat{x}$.
- Each expanded expression form, e, maps onto an unexpanded expression form $\mathcal{U}(e)$ as follows:

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

- 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:

$$\begin{aligned}
\mathcal{U}(x) &= \widehat{x} \\
\mathcal{U}(\texttt{wildp}) &= \texttt{uwildp} \\
\mathcal{U}(\texttt{foldp(}p\texttt{)}) &= \texttt{ufoldp(}\mathcal{U}(p)\texttt{)}
\end{aligned}$$

$$\mathcal{U}(\texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L})) = \texttt{utplp}[L](\{i \hookrightarrow \mathcal{U}(p_i)\}_{i \in L})$$
$$\mathcal{U}(\texttt{inp}[\ell](p)) = \texttt{uinp}[\ell](\mathcal{U}(p))$$

The only unexpanded forms that do not correspond to expanded forms are the unexpanded expression forms for ueTSM definition, ueTSM application and upTSM definition, and the unexpanded pattern form for upTSM application. The forms related to upTSMs are highlighted in gray in Figure 4.3.

4.2.5 Typed Expansion

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

Judgement Form	**Description**
$\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau$ type	$\hat{\tau}$ is well-formed and has expansion τ assuming $\hat{\Delta}$
$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau$	$\hat{e}$ has expansion e and type τ under ueTSM context $\hat{\Psi}$ and upTSM context $\hat{\Phi}$ assuming $\hat{\Delta}$ and $\hat{\Gamma}$
$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \rightsquigarrow r : \tau \Mapsto \tau'$	$\hat{r}$ has expansion r and takes values of type τ to values of type τ' under $\hat{\Psi}$ and $\hat{\Phi}$ assuming $\hat{\Delta}$ and $\hat{\Gamma}$
$\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\mid \hat{\Upsilon}$	$\hat{p}$ has expansion p and type τ and generates hypotheses Υ under upTSM context $\hat{\Phi}$ assuming Δ

Type Expansion

Unexpanded type formation contexts, $\hat{\Delta}$, were defined in Sec. 3.2.5. The *type expansion judgement*, $\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau$ type, is inductively defined by Rules (3.3).

Typed Expression, Rule and Pattern Expansion

Unexpanded typing contexts, $\hat{\Gamma}$, were defined in Sec. 3.2.5. *Unexpanded pattern typing contexts*, $\hat{\Upsilon}$, are defined identically to unexpanded typing contexts (i.e. we only use a distinct metavariable to emphasize their distinct roles in the judgements above).

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 (4.4) and Rule (4.5), respectively, and the *typed pattern expansion judgement*, $\Delta \vdash_{\Phi} \hat{p} \rightsquigarrow p : \tau \dashv\mid \Upsilon$, is inductively defined by Rules (4.6) as follows.

Shared Forms Rules (4.4a) through (4.4k) define typed expansion of unexpanded expressions of shared form. The first five of these rules are shown below:

$$\frac{}{\hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{x} \rightsquigarrow x : \tau} \tag{4.4a}$$

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \text{ type} \qquad \hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{ulam\{}\hat{\tau}\mathtt{\}(}\hat{x}.\hat{e}\mathtt{)} \rightsquigarrow \mathtt{lam\{}\tau\mathtt{\}(}x.e\mathtt{)} : \mathtt{parr(}\tau;\tau'\mathtt{)}} \tag{4.4b}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_1 \rightsquigarrow e_1 : \mathtt{parr(}\tau;\tau'\mathtt{)} \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_2 \rightsquigarrow e_2 : \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uap(}\hat{e}_1;\hat{e}_2\mathtt{)} \rightsquigarrow \mathtt{ap(}e_1;e_2\mathtt{)} : \tau'} \tag{4.4c}$$

$$\frac{\hat{\Delta}, \hat{t} \rightsquigarrow t \text{ type}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{utlam(}\hat{t}.\hat{e}\mathtt{)} \rightsquigarrow \mathtt{tlam(}t.e\mathtt{)} : \mathtt{all(}t.\tau\mathtt{)}} \tag{4.4d}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \mathtt{all(}t.\tau\mathtt{)} \qquad \hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau' \text{ type}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{utap\{}\hat{\tau}'\mathtt{\}(}\hat{e}\mathtt{)} \rightsquigarrow \mathtt{tap\{}\tau'\mathtt{\}(}e\mathtt{)} : [\tau'/t]\tau} \tag{4.4e}$$

These rules are similar to Rules (3.4a) through (3.4e). In particular, in both sets of rules, the unexpanded and expanded expression forms in the conclusion correspond, and the premises correspond to those of the typing rule for the expanded expression form – here, Rules (4.1a) through (4.1e), respectively. In particular, each type expansion premise in each rule above corresponds to a type formation premise in the corresponding typing rule, and each typed expression expansion premise in each rule above corresponds to a typing premise in 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 ueTSM context, $\hat{\Psi}$, and now also the upTSM context, $\hat{\Phi}$, pass opaquely through these rules.

Rule (4.4k), below, handles unexpanded match expressions and corresponds in the same way to Rule (4.1k).

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau \qquad \hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau' \text{ type} \qquad \{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \rightsquigarrow r_i : \tau \Rightarrow \tau'\}_{1 \leq i \leq n}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{umatch}[n]\mathtt{\{}\hat{\tau}'\mathtt{\}(}\hat{e};\{\hat{r}_i\}_{1 \leq i \leq n}\mathtt{)} \rightsquigarrow \mathtt{match}[n]\mathtt{\{}\tau'\mathtt{\}(}e;\{r_i\}_{1 \leq i \leq n}\mathtt{)} : \tau'} \tag{4.4k}$$

We can express this scheme more precisely with the following rule transformation. For each rule in Rules (4.1),

$$\frac{J_1 \qquad \cdots \qquad J_k}{J}$$

the corresponding typed expansion rule is

$$\frac{\mathcal{U}(J_1) \qquad \cdots \qquad \mathcal{U}(J_k)}{\mathcal{U}(J)}$$

where

$$\begin{aligned}
\mathcal{U}(\Delta \vdash \tau \text{ type}) &= \mathcal{U}(\Delta) \vdash \mathcal{U}(\tau) \rightsquigarrow \tau \text{ type} \\
\mathcal{U}(\Gamma\ \Delta \vdash e : \tau) &= \mathcal{U}(\Gamma)\ \mathcal{U}(\Delta) \vdash_{\hat{\Psi};\hat{\Phi}} \mathcal{U}(e) \rightsquigarrow e : \tau \\
\mathcal{U}(\Gamma\ \Delta \vdash r : \tau \Rightarrow \tau') &= \mathcal{U}(\Gamma)\ \mathcal{U}(\Delta) \vdash_{\hat{\Psi};\hat{\Phi}} \mathcal{U}(r) \rightsquigarrow r : \tau \Rightarrow \tau' \\
\mathcal{U}(\{J_i\}_{i \in L}) &= \{\mathcal{U}(J_i)\}_{i \in L}
\end{aligned}$$

and where $\mathcal{U}(\Delta)$, $\mathcal{U}(\Gamma)$ and $\mathcal{U}(\tau)$ are defined as in Sec. 3.2.5 and:

- $\mathcal{U}(e)$ is defined as follows
  - When e is of definite form, $\mathcal{U}(e)$ is defined as in Sec. 4.2.4.
  - When e is of indefinite form, $\mathcal{U}(e)$ is a uniquely corresponding metavariable of sort UExp also of indefinite form. For example, $\mathcal{U}(e_1) = \hat{e}_1$ and $\mathcal{U}(e_2) = \hat{e}_2$.
- $\mathcal{U}(r)$ is defined as follows:
  - When r is of definite form, $\mathcal{U}(r)$ is defined as in Sec. 4.2.4.
  - When e is of indefinite form, $\mathcal{U}(r)$ is a uniquely corresponding metavariable of sort URule also of indefinite form.

It is instructive to use this rule transformation to generate Rules (4.4a) through (4.4e) and Rule (4.4k) above. We omit the remaining rules generated by this transformation, i.e. Rules (4.4d) through (4.4j).

The typed rule expansion judgement is defined by Rule (4.5), below.

$$\frac{\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid\! \langle \mathcal{G}'; \Upsilon \rangle \qquad \langle \mathcal{D}; \Delta \rangle \ \langle \mathcal{G} \uplus \mathcal{G}'; \Gamma \cup \Upsilon \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau'}{\langle \mathcal{D}; \Delta \rangle \ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{urule}(\hat{p}.\hat{e}) \rightsquigarrow \mathtt{rule}(p.e) : \tau \Rightarrow \tau'} \tag{4.5}$$

As in the typed expression expansion judgements, the unexpanded and expanded forms in the conclusion of the rule above correspond. The premises correspond to those of Rule (4.2). In particular, the typed pattern expansion premise in the rule above corresponds to the pattern typing premise of Rule (4.2) and the typed expression expansion premise in the rule above corresponds to the typing premise of Rule (4.2). Because unexpanded terms bind expression sigils, which are given meaning by expansion to variables, the pattern typing rules must generate both a sigil expansion context, $\mathcal{G}'$, and a pattern typing context, Υ. In the second premise, we update the "incoming" sigil expansion context, $\mathcal{G}$, with the new sigil expansions, $\mathcal{G}'$, and correspondingly, extend the "incoming" typing context, Γ, with the new hypotheses, Υ.

Rules (4.6a) through (4.6e), below, define typed expansion of unexpanded patterns of shared form.

$$\frac{}{\Delta \vdash_{\hat{\Phi}} \hat{x} \rightsquigarrow x : \tau \dashv\!\mid\! \langle \hat{x} \rightsquigarrow x; x : \tau \rangle} \tag{4.6a}$$

$$\frac{}{\Delta \vdash_{\hat{\Phi}} \mathtt{uwildp} \rightsquigarrow \mathtt{wildp} : \tau \dashv\!\mid\! \langle \emptyset; \emptyset \rangle} \tag{4.6b}$$

$$\frac{\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : [\mathtt{rec}(t.\tau)/t]\tau \dashv\!\mid\! \hat{\Upsilon}}{\Delta \vdash_{\hat{\Phi}} \mathtt{ufoldp}(\hat{p}) \rightsquigarrow \mathtt{foldp}(p) : \mathtt{rec}(t.\tau) \dashv\!\mid\! \hat{\Upsilon}} \tag{4.6c}$$

$$\frac{\{\Delta \vdash_{\hat{\Phi}} \hat{p}_i \rightsquigarrow p_i : \tau_i \dashv\!\mid\! \hat{\Upsilon}_i\}_{i \in L}}{\left(\begin{array}{c} \Delta \vdash_{\hat{\Phi}} \mathtt{utplp}[L](\{i \hookrightarrow \hat{p}_i\}_{i \in L}) \\ \rightsquigarrow \\ \mathtt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\!\mid\! \cup_{i \in L} \hat{\Upsilon}_i \end{array}\right)} \tag{4.6d}$$

$$\frac{\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid\! \hat{\Upsilon}}{\Delta \vdash_{\hat{\Phi}} \mathtt{uinp}[\ell](\hat{p}) \rightsquigarrow \mathtt{inp}[\ell](p) : \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\!\mid\! \hat{\Upsilon}} \tag{4.6e}$$

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 (4.3a) through (4.3e), respectively. The upTSM context, $\hat{\Phi}$, passes through these rules opaquely. In Rule (4.6d), the conclusion of the rule collects all of the sigil expansions and hypotheses generated by the subpatterns. We define $\hat{\Upsilon}_i$ as shorthand for $\langle \mathcal{G}_i; \Upsilon_i \rangle$ and $\cup_{i \in L} \hat{\Upsilon}_i$ as shorthand for

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

By the definition of iterated extension of finite functions, we implicitly have that no sigils 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

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

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 $\mathcal{U}(\Upsilon)$ maps Υ to an unexpanded typing context as follows:

$$\begin{aligned} \mathcal{U}(\emptyset) &= \langle \emptyset; \emptyset \rangle \\ \mathcal{U}(\Upsilon, x : \tau) &= \mathcal{U}(\Upsilon), \widehat{x} \rightsquigarrow x : \tau \\ \mathcal{U}(\cup_{i \in L} \Upsilon_i) &= \cup_{i \in L} \mathcal{U}(\Upsilon_i) \end{aligned}$$

Lemma 4.12 (Pattern Expressibility)**.** *If* $\Delta \vdash p : \tau \dashv\mid \Upsilon$ *then* $\Delta \vdash_{\hat{\Phi}} \mathcal{U}(p) \rightsquigarrow p : \tau \dashv\mid \mathcal{U}(\Upsilon)$.

Proof. By rule induction over Rules (4.3), using the definitions of $\mathcal{U}(\Upsilon)$ and $\mathcal{U}(p)$ given above. In each case, we can apply the IH to or over each premise, then apply the corresponding rule in Rules (4.6). □

We can now establish the Expressibility Theorem – that each well-typed expanded expression, e, can be expressed as an unexpanded expression, $\hat{e}$, and assigned the same type under the corresponding contexts.

Theorem 4.13 (Expressibility)**.** *Both of the following hold:*

1. *If* $\Delta\ \Gamma \vdash e : \tau$ *then* $\mathcal{U}(\Delta)\ \mathcal{U}(\Gamma) \vdash_{\hat{\Psi}; \hat{\Phi}} \mathcal{U}(e) \rightsquigarrow e : \tau$.
2. *If* $\Delta\ \Gamma \vdash r : \tau \Rrightarrow \tau'$ *then* $\mathcal{U}(\Delta)\ \mathcal{U}(\Gamma) \vdash_{\hat{\Psi}; \hat{\Phi}} \mathcal{U}(r) \rightsquigarrow r : \tau \Rrightarrow \tau'$.

Proof. By mutual rule induction over Rules (4.1) and Rule (4.2).

For part 1, we induct on the assumption. The rule transformation defined above guarantees that this part holds by its construction. In particular, in each case, we can apply Lemma 3.13 to or over each type formation premise, the IH (part 1) to or over each typing premise, the IH (part 2) over each rule typing premise, then apply the corresponding rule in Rules (4.4).

For part 2, we induct on the assumption. There is only one case:

Case (4.2)**.** We have:

(1) $r = \texttt{rule}(p.e)$ by assumption
(2) $\Delta \vdash p : \tau \dashv\!\mid \Upsilon$ by assumption
(3) $\Delta\ \Gamma \cup \Upsilon \vdash e : \tau'$ by assumption
(4) $\mathcal{U}(\Gamma) = \langle \mathcal{G}; \Gamma \rangle$, for some $\mathcal{G}$ by definition of $\mathcal{U}(\Gamma)$
(5) $\mathcal{U}(\Upsilon) = \langle \mathcal{G}'; \Upsilon \rangle$, for some $\mathcal{G}'$ by definition of $\mathcal{U}(\Upsilon)$
(6) $\mathcal{U}(\Gamma \cup \Upsilon) = \langle \mathcal{G} \uplus \mathcal{G}'; \Gamma \cup \Upsilon \rangle$ by definition of $\mathcal{U}(\Upsilon)$
(7) $\mathcal{U}(r) = \texttt{urule}(\mathcal{U}(p).\mathcal{U}(e))$ by definition of $\mathcal{U}(r)$
(8) $\Delta \vdash_{\hat{\Phi}} \mathcal{U}(p) \rightsquigarrow p : \tau \dashv\!\mid \langle \mathcal{G}'; \Upsilon \rangle$ by Lemma 4.12 on (2)
(9) $\hat{\Delta}\ \langle \mathcal{G} \uplus \mathcal{G}'; \Gamma \cup \Upsilon \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \mathcal{U}(e) \rightsquigarrow e : \tau'$ by IH, part 1 on (3)
(10) $\mathcal{U}(\Delta)\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{urule}(\mathcal{U}(p).\mathcal{U}(e)) \rightsquigarrow \texttt{rule}(p.e) : \tau \Rightarrow \tau'$ by Rule (4.5) on (8) and (9)

□

ueTSM Definition and Application Rules (4.4l) and (4.4m) define typed expansion of ueTSM definitions and ueTSM application, respectively.

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \emptyset\,\emptyset \vdash e_{\mathsf{parse}} : \texttt{parr}(\texttt{Body};\texttt{ParseResultExp}) \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uetsm}(\tau;e_{\mathsf{parse}});\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for expressions}\ \{e_{\mathsf{parse}}\}\ \texttt{in}\ \hat{e} \rightsquigarrow e : \tau'} \tag{4.4l}$$

$$\frac{b \downarrow e_{\mathsf{body}} \qquad e_{\mathsf{parse}}(e_{\mathsf{body}}) \Downarrow \texttt{inj}\ \texttt{Success} \cdot e_{\mathsf{cand}} \qquad e_{\mathsf{cand}} \uparrow_{\mathsf{CEExp}} \grave{e} \qquad \emptyset\,\emptyset \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uetsm}(\tau;e_{\mathsf{parse}});\hat{\Phi};b} \grave{e} \rightsquigarrow e : \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uetsm}(\tau;e_{\mathsf{parse}});\hat{\Phi}} \hat{a}\ /b/ \rightsquigarrow e : \tau} \tag{4.4m}$$

These rules are nearly identical to Rules (3.4l) and (3.4m), respectively, differing only in that the upTSM context, $\hat{\Phi}$, passes through them opaquely. The premises of these rules, and the following auxiliary definitions and conditions, can be understood as described in Sec. 3.2.6 and 3.2.7.

ueTSM contexts, $\hat{\Psi}$, are of the form $\langle \mathcal{A}; \Psi \rangle$, where $\mathcal{A}$ is a *TSM naming context* and Ψ is a *ueTSM definition context*. TSM naming contexts were defined in Sec. 3.2.6. A *ueTSM definition context*, Ψ, is a finite function mapping each symbol $a \in \mathrm{dom}(\Psi)$ to an *expanded ueTSM definition*, $a \hookrightarrow \texttt{uetsm}(\tau; e_{\mathsf{parse}})$, where τ is the ueTSM's type annotation, and e_{parse} is its parse function. We write $\Psi, a \hookrightarrow \texttt{uetsm}(\tau; e_{\mathsf{parse}})$ when $a \notin \mathrm{dom}(\Psi)$ for the extension of Ψ that maps a to $a \hookrightarrow \texttt{uetsm}(\tau; e_{\mathsf{parse}})$. We write $\Delta \vdash \Psi\ \mathsf{ueTSMs}$ when all the type annotations in Ψ are well-formed assuming Δ, and the parse functions in Ψ are closed and of type $\texttt{Body} \rightharpoonup \texttt{ParseResultExp}$.

Definition 4.14 (ueTSM Definition Context Formation)**.** $\Delta \vdash \Psi\ \mathsf{ueTSMs}$ *iff for each* $\hat{a} \hookrightarrow \texttt{uetsm}(\tau; e_{parse}) \in \Psi$, *we have* $\Delta \vdash \tau\ \mathsf{type}$ *and* $\emptyset\,\emptyset \vdash e_{parse} : \texttt{Body} \rightharpoonup \texttt{ParseResultExp}$.

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

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

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* $b \downarrow e_{\text{body}}$. An inverse mapping is defined by the *body decoding judgement* $e_{\text{body}} \uparrow b$.

Judgement Form	**Description**
$b \downarrow e$	b has encoding e
$e \uparrow b$	e has decoding b

The following condition establishes an isomorphism between literal bodies and values of type `Body` mediated by the judgements above.

Condition 4.15 (Body Isomorphism)**.** *All of the following must hold:*

1. *For every literal body b, we have that $b \downarrow e_{body}$ for some e_{body} such that $\vdash e_{body} : \texttt{Body}$ and e_{body} val.*
2. *If $\vdash e_{body} : \texttt{Body}$ and e_{body} val then $e_{body} \uparrow b$ for some b.*
3. *If $b \downarrow e_{body}$ then $e_{body} \uparrow b$.*
4. *If $\vdash e_{body} : \texttt{Body}$ and e_{body} val and $e_{body} \uparrow b$ then $b \downarrow e_{body}$.*
5. *If $b \downarrow e_{body}$ and $b \downarrow e'_{body}$ then $e_{body} = e'_{body}$.*
6. *If $\vdash e_{body} : \texttt{Body}$ and e_{body} val and $e_{body} \uparrow b$ and $e_{body} \uparrow b'$ then $b = b'$.*

`ParseResultExp` abbreviates a labeled sum type that distinguishes successful parses from parse errors:

$$\texttt{ParseResultExp} \triangleq [\texttt{Success} \hookrightarrow \texttt{CEExp}, \texttt{ParseError} \hookrightarrow \langle\rangle]$$

The type abbreviated `CEExp` classifies encodings of *candidate expansion expressions* (or *ce-expressions*), $\grave{e}$ (pronounced "grave e"). The syntax of ce-expressions will be described in Sec. 4.2.8. The mapping from ce-expressions to values of type `CEExp` is defined by the *ce-expression encoding judgement*, $\grave{e} \downarrow_{\mathsf{CEExp}} e$. An inverse mapping is defined by the *ce-expression decoding judgement*, $e \uparrow_{\mathsf{CEExp}} \grave{e}$.

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

The following condition establishes an isomorphism between values of type `CEExp` and ce-expressions.

Condition 4.16 (Candidate Expansion Expression Isomorphism)**.** *All of the following hold:*

1. *For every $\grave{e}$, we have $\grave{e} \downarrow_{\mathsf{CEExp}} e_{cand}$ for some e_{cand} such that $\vdash e_{cand} : \texttt{CEExp}$ and e_{cand} val.*
2. *If $\vdash e_{cand} : \texttt{CEExp}$ and e_{cand} val then $e_{cand} \uparrow_{\mathsf{CEExp}} \grave{e}$ for some $\grave{e}$.*
3. *If $\grave{e} \downarrow_{\mathsf{CEExp}} e_{cand}$ then $e_{cand} \uparrow_{\mathsf{CEExp}} \grave{e}$.*
4. *If $\vdash e_{cand} : \texttt{CEExp}$ and e_{cand} val and $e_{cand} \uparrow_{\mathsf{CEExp}} \grave{e}$ then $\grave{e} \downarrow_{\mathsf{CEExp}} e_{cand}$.*
5. *If $\grave{e} \downarrow_{\mathsf{CEExp}} e_{cand}$ and $\grave{e} \downarrow_{\mathsf{CEExp}} e'_{cand}$ then $e_{cand} = e'_{cand}$.*
6. *If $\vdash e_{cand} : \texttt{CEExp}$ and e_{cand} val and $e_{cand} \uparrow_{\mathsf{CEExp}} \grave{e}$ and $e_{cand} \uparrow_{\mathsf{CEExp}} \grave{e}'$ then $\grave{e} = \grave{e}'$.*

upTSM Definition and Application Rules (4.4n) and (4.6f) define upTSM definition and application, and are defined in the next two subsections, respectively.

4.2.6 upTSM Definition

The stylized upTSM definition form is

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

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

Rule (4.4n) defines typed expansion of upTSM definitions:

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \text{type} \qquad \varnothing\,\varnothing \vdash e_{\text{parse}} : \texttt{Body} \rightharpoonup \texttt{ParseResultPat} \qquad \hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uptsm}(\tau;e_{\text{parse}})} \hat{e} \rightsquigarrow e : \tau'}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for patterns}\ \{e_{\text{parse}}\}\ \texttt{in}\ \hat{e} \rightsquigarrow e : \tau'} \tag{4.4n}$$

This rule is similar to Rule (4.4l), which governs ueTSM definitions. The premises of this rule can be understood as follows, in order:

1. The first premise ensures that the unexpanded type annotation is well-formed and expands it to produce the *type annotation*, τ.
2. The second premise checks that the parse function, e_{parse}, is closed and of type

   $$\texttt{Body} \rightharpoonup \texttt{ParseResultExp}$$

   The type abbreviated `Body` is characterized above.

   `ParseResultPat`, like `ParseResultExp` above, abbreviates a labeled sum type that distinguishes successful parses from parse errors:

   $$\texttt{ParseResultPat} \triangleq [\texttt{Success} \hookrightarrow \texttt{CEPat}, \texttt{ParseError} \hookrightarrow \langle\rangle]$$

   The type abbreviated `CEPat` classifies encodings of *candidate expansion patterns* (or *ce-patterns*), $\grave{p}$. The syntax of ce-patterns will be described in Sec. 4.2.8. The mapping from ce-patterns to values of type `CEPat` is defined by the *ce-pattern encoding judgement*, $\grave{p} \downarrow_{\texttt{CEPat}} e$. An inverse mapping is defined by the *ce-pattern decoding judgement*, $e \uparrow_{\texttt{CEPat}} \grave{p}$.

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

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

   Condition 4.17 (Candidate Expansion Pattern Isomorphism)**.** *All of the following must hold:*

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

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

(c) *If* $\grave{p} \downarrow_{\texttt{CEPat}} e_{cand}$ *then* $e_{cand} \uparrow_{\texttt{CEPat}} \grave{p}$.

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

(e) *If* $\grave{p} \downarrow_{\texttt{CEPat}} e_{cand}$ *and* $\grave{p} \downarrow_{\texttt{CEPat}} e'_{cand}$ *then* $e_{cand} = e'_{cand}$.

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

3. The final premise of Rule (4.4n) extends the upTSM context, $\hat{\Phi}$, with the newly determined upTSM definition, and proceeds to assign a type, τ', and expansion, e, to $\hat{e}$. The conclusion of Rule (4.4n) assigns this type and expansion to the upTSM definition as a whole.

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

   A *upTSM definition context*, Φ, is a finite function mapping each symbol $a \in \mathrm{dom}(\Phi)$ to an *expanded upTSM definition*, $a \hookrightarrow \texttt{uptsm}(\tau; e_{\text{parse}})$, where τ is the upTSM's type annotation, and e_{parse} is its parse function. We write $\Phi, a \hookrightarrow \texttt{uptsm}(\tau; e_{\text{parse}})$ when $a \notin \mathrm{dom}(\Phi)$ for the extension of Φ that maps a to $a \hookrightarrow \texttt{uptsm}(\tau; e_{\text{parse}})$. We write $\Delta \vdash \Phi$ upTSMs when all the type annotations in Φ are well-formed assuming Δ, and the parse functions in Φ are closed and of type $\texttt{Body} \rightharpoonup \texttt{ParseResultPat}$.

   Definition 4.18 (upTSM Definition Context Formation)**.** $\Delta \vdash \Phi$ upTSMs *iff for each* $a \hookrightarrow \texttt{uptsm}(\tau; e_{parse}) \in \Phi$, *we have* $\Delta \vdash \tau$ type *and* $\varnothing\,\varnothing \vdash e_{parse} : \texttt{Body} \rightharpoonup \texttt{ParseResultPat}$.

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

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

   and $\hat{\Phi} \cup \hat{\Phi}'$ when $\hat{\Phi} = \langle \mathcal{A}; \Phi \rangle$ and $\hat{\Phi}' = \langle \mathcal{A}'; \Phi' \rangle$ as an abbreviation of

$$\langle \mathcal{A} \cup \mathcal{A}'; \Phi \cup \Phi' \rangle$$

4.2.7 upTSM Application

The stylized unexpanded pattern form for applying a upTSM 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 ueTSM application, differing in that appears within the syntax of unexpanded patterns, $\hat{p}$, rather than unexpanded expressions, $\hat{e}$. The corresponding operational form is $\texttt{uapuptsm}[b]\texttt{[}\hat{a}\texttt{]}$.

Rule (4.6f), below, governs upTSM application.

$$\frac{b \downarrow e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj Success} \cdot e_{\text{cand}} \qquad e_{\text{cand}} \uparrow_{\texttt{CEPat}} \grave{p} \qquad \vdash^{\Delta; \hat{\Phi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{uptsm}(\tau; e_{\text{parse}}); b} \grave{p} \rightsquigarrow p : \tau \dashv\!\!\mid^{\hat{Y}}}{\Delta \vdash_{\hat{\Phi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{uptsm}(\tau; e_{\text{parse}})} \hat{a}\ /b/ \rightsquigarrow p : \tau \dashv\!\!\mid \hat{Y}} \tag{4.6f}$$

This rule is similar to Rule (4.4m), which governs ueTSM application. Its premises can be understood as follows, in order:

1. The first premise determines the encoding of the literal body, e_{body}.
2. The second premise applies the parse function e_{parse} to e_{body}. If parsing succeeds, i.e. a value of the (stylized) form $\texttt{inj Success} \cdot e_{\text{cand}}$ results from evaluation, then e_{cand} will be a value of type `CEPat` (assuming a well-formed upTSM context, by application of Assumption 4.9). We call e_{cand} the *encoding of the candidate expansion*.
3. The third premise decodes the encoding of the candidate expansion to produce *candidate expansion*, $\grave{p}$.
4. The final premise *validates* the candidate expansion and simultaneously generates the final expansion, p, and assumptions, Υ. This is the topic of Sec. 4.2.9.

4.2.8 Syntax of Candidate Expansions

Figure 4.4 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. We refer to these as the *shared forms*. In particular:

- Each type form maps onto a ce-type form according to the metafunction $\mathcal{C}(\tau)$, defined in Sec. 3.2.8.
- Each expanded expression form maps onto a ce-expression form according to the metafunction $\mathcal{C}(e)$, defined as follows:

$$
\begin{aligned}
\mathcal{C}(x) &= x \\
\mathcal{C}(\texttt{lam\{}\tau\texttt{\}(}x.e\texttt{)}) &= \texttt{celam\{}\mathcal{C}(\tau)\texttt{\}(}x.\mathcal{C}(e)\texttt{)} \\
\mathcal{C}(\texttt{ap(}e_1\texttt{;}e_2\texttt{)}) &= \texttt{ceap(}\mathcal{C}(e_1)\texttt{;}\mathcal{C}(e_2)\texttt{)} \\
\mathcal{C}(\texttt{tlam(}t.e\texttt{)}) &= \texttt{cetlam(}t.\mathcal{C}(e)\texttt{)} \\
\mathcal{C}(\texttt{tap\{}\tau\texttt{\}(}e\texttt{)}) &= \texttt{cetap\{}\mathcal{C}(\tau)\texttt{\}(}\mathcal{C}(e)\texttt{)} \\
\mathcal{C}(\texttt{fold\{}t.\tau\texttt{\}(}e\texttt{)}) &= \texttt{cefold\{}t.\mathcal{C}(\tau)\texttt{\}(}\mathcal{C}(e)\texttt{)} \\
\mathcal{C}(\texttt{unfold(}e\texttt{)}) &= \texttt{ceunfold(}\mathcal{C}(e)\texttt{)} \\
\mathcal{C}(\texttt{tpl}[L]\texttt{(}\{i \hookrightarrow e_i\}_{i \in L}\texttt{)}) &= \texttt{cetpl}[L]\texttt{(}\{i \hookrightarrow \mathcal{C}(e_i)\}_{i \in L}\texttt{)} \\
\mathcal{C}(\texttt{in}[L]\texttt{[}\ell\texttt{]\{}\{i \hookrightarrow \tau_i\}_{i \in L}\texttt{\}(}e\texttt{)}) &= \texttt{cein}[L]\texttt{[}\ell\texttt{]\{}\{i \hookrightarrow \mathcal{C}(\tau_i)\}_{i \in L}\texttt{\}(}\mathcal{C}(e)\texttt{)} \\
\mathcal{C}(\texttt{match}[n]\texttt{\{}\tau\texttt{\}(}e\texttt{;}\{r_i\}_{1 \leq i \leq n}\texttt{)}) &= \texttt{cematch}[n]\texttt{\{}\mathcal{C}(\tau)\texttt{\}(}\mathcal{C}(e)\texttt{;}\{\mathcal{C}(r_i)\}_{1 \leq i \leq n}\texttt{)}
\end{aligned}
$$

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

$$\mathcal{C}(\texttt{rule(}p.e\texttt{)}) = \texttt{cerule(}p.\mathcal{C}(e)\texttt{)}$$

Sort			Operational Form	Stylized Form	Description
CETyp	$\grave{\tau}$	::=	t	t	variable
			$\texttt{ceparr}(\grave{\tau};\grave{\tau})$	$\grave{\tau} \rightharpoonup \grave{\tau}$	partial function
			$\texttt{ceall}(t.\grave{\tau})$	$\forall t.\grave{\tau}$	polymorphic
			$\texttt{cerec}(t.\grave{\tau})$	$\mu t.\grave{\tau}$	recursive
			$\texttt{ceprod}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i\in L})$	$\langle\{i \hookrightarrow \grave{\tau}_i\}_{i\in L}\rangle$	labeled product
			$\texttt{cesum}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i\in L})$	$[\{i \hookrightarrow \grave{\tau}_i\}_{i\in L}]$	labeled sum
			$\texttt{cesplicedt}[m;n]$	$\texttt{splicedt}\langle m,n\rangle$	spliced
CEExp	$\grave{e}$	::=	x	x	variable
			$\texttt{celam}\{\grave{\tau}\}(x.\grave{e})$	$\lambda x{:}\grave{\tau}.\grave{e}$	abstraction
			$\texttt{ceap}(\grave{e};\grave{e})$	$\grave{e}(\grave{e})$	application
			$\texttt{cetlam}(t.\grave{e})$	$\Lambda t.\grave{e}$	type abstraction
			$\texttt{cetap}\{\grave{\tau}\}(\grave{e})$	$\grave{e}[\grave{\tau}]$	type application
			$\texttt{cefold}\{t.\grave{\tau}\}(\grave{e})$	$\texttt{fold}(\grave{e})$	fold
			$\texttt{ceunfold}(\grave{e})$	$\texttt{unfold}(\grave{e})$	unfold
			$\texttt{cetpl}[L](\{i \hookrightarrow \grave{e}_i\}_{i\in L})$	$\langle\{i \hookrightarrow \grave{e}_i\}_{i\in L}\rangle$	labeled tuple
			$\texttt{cepr}[\ell](\grave{e})$	$\grave{e}\cdot\ell$	projection
			$\texttt{cein}[L][\ell]\{\{i \hookrightarrow \grave{\tau}_i\}_{i\in L}\}(\grave{e})$	$\texttt{inj}\ \ell\cdot\grave{e}$	injection
			$\texttt{cematch}[n]\{\grave{\tau}\}(\grave{e};\{\grave{r}_i\}_{1\le i\le n})$	$\texttt{match}\ \grave{e}\ \{\grave{r}_i\}_{1\le i\le n}$	match
			$\texttt{cesplicede}[m;n]$	$\texttt{splicede}\langle m,n\rangle$	spliced
CERule	$\grave{r}$	::=	$\texttt{cerule}(p.\grave{e})$	$p \Rightarrow \grave{e}$	rule
CEPat	$\grave{p}$	::=	$\texttt{cewildp}$	$\_$	wildcard pattern
			$\texttt{cefoldp}(p)$	$\texttt{fold}(p)$	fold pattern
			$\texttt{cetplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i\in L})$	$\langle\{i \hookrightarrow \grave{p}_i\}_{i\in L}\rangle$	labeled tuple pattern
			$\texttt{ceinp}[\ell](\grave{p})$	$\texttt{inj}\ \ell\cdot\grave{p}$	injection pattern
			$\texttt{cesplicedp}[m;n]$	$\texttt{splicedp}\langle m,n\rangle$	spliced

Figure 4.4: Abstract syntax of candidate expansion types, expressions, rules and patterns in $\mathsf{miniVerse}_{\mathsf{U}}$. Candidate expansion terms are identified up to α-equivalence.

Notice that ce-rules bind expanded patterns, not ce-patterns. This is because ce-rules appear in ce-expressions, which are generated by ueTSMs. It would not be sensible for a ueTSM to splice a pattern out of a literal body.

- Each expanded pattern form, except for the variable pattern form, maps onto a ce-pattern form according to the metafunction $\mathcal{C}(p)$, defined as follows:

$$\begin{aligned}
\mathcal{C}(\texttt{wildp}) &= \texttt{cewildp} \\
\mathcal{C}(\texttt{foldp(}p\texttt{)}) &= \texttt{cefoldp(}\mathcal{C}(p)\texttt{)} \\
\mathcal{C}(\texttt{tplp}[L]\texttt{(}\{i \hookrightarrow p_i\}_{i \in L}\texttt{)}) &= \texttt{cetplp}[L]\texttt{(}\{i \hookrightarrow \mathcal{C}(p_i)\}_{i \in L}\texttt{)} \\
\mathcal{C}(\texttt{inp[}\ell\texttt{](}p\texttt{)}) &= \texttt{ceinp[}\ell\texttt{](}\mathcal{C}(p)\texttt{)}
\end{aligned}$$

There are three other candidate expansion forms: a ce-type form for *references to spliced unexpanded types*, $\texttt{cesplicedt}[m; n]$, a ce-expression form for *references to spliced unexpanded expressions*, $\texttt{cesplicede}[m; n]$, and, highlighted in gray in Figure 4.4, a ce-pattern form for *references to spliced unexpanded patterns*, $\texttt{cesplicedp}[m; n]$.

4.2.9 Candidate Expansion Validation

The *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 $\mathbb{T}$
$\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau$	$\grave{e}$ has expansion e and type τ assuming Δ and Γ and expression splicing scene $\mathbb{E}$
$\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r} \rightsquigarrow r : \tau \Rrightarrow \tau'$	$\grave{r}$ has expansion r and takes values of type τ to values of type τ' assuming Δ and Γ and expression splicing scene $\mathbb{E}$
$\vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\hat{Y}}$	$\grave{p}$ expands to p and matches values of type τ generating assumptions $\hat{Y}$ 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 (4.4m) and (4.6f)), 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 (3.5), which were defined in Sec. 3.2.9.

Candidate Expansion Expression and Rule Validation

The *ce-expression validation judgement*, $\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau$, and the *ce-rule validation judgement*, $\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r} \rightsquigarrow r : \tau \Mapsto \tau'$, are defined mutually inductively with Rules (4.4) and Rule (4.5) by Rules (4.7) and Rule (4.8), respectively, as follows.

Rules (4.7) define ce-expression validation and consist of the following rules:

- Rules written identically to Rules (3.6a) through (3.6j). We will refer to these as Rules (4.7a) through (4.7j).
- The following rule for match ce-expressions:

$$\frac{\begin{array}{c}\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau \qquad \Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau}' \rightsquigarrow \tau' \ \mathsf{type} \\ \{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r}_i \rightsquigarrow r_i : \tau \Mapsto \tau'\}_{1 \leq i \leq n}\end{array}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{cematch}[n]\{\grave{\tau}'\}(\grave{e}; \{\grave{r}_i\}_{1 \leq i \leq n}) \rightsquigarrow \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \leq i \leq n}) : \tau'} \tag{4.7k}$$

- The following rule for references to spliced unexpanded expressions, which can be understood as described in Sec. 3.2.9.

$$\frac{\begin{array}{c}\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e} \qquad \langle \mathcal{D}; \Delta_{\mathrm{app}} \rangle\ \langle \mathcal{G}; \Gamma_{\mathrm{app}} \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e : \tau \\ \Delta \cap \Delta_{\mathrm{app}} = \emptyset \qquad \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma_{\mathrm{app}}) = \emptyset\end{array}}{\Delta\ \Gamma \vdash^{\langle \mathcal{D}; \Delta_{\mathrm{app}} \rangle; \langle \mathcal{G}; \Gamma_{\mathrm{app}} \rangle; \hat{\Psi}; \hat{\Phi}; b} \mathtt{cesplicede}[m; n] \rightsquigarrow e : \tau} \tag{4.7l}$$

Rule (4.8) defines ce-rule validation and is defined as follows:

$$\frac{\Delta \vdash p : \tau \dashv\!| \mathcal{Y} \qquad \Delta\ \Gamma \cup \mathcal{Y} \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{cerule}(p.\grave{e}) \rightsquigarrow \mathtt{rule}(p.e) : \tau \Mapsto \tau'} \tag{4.8}$$

The following lemma establishes that each well-typed expanded expression, e, can be expressed as a valid ce-expression, $\mathcal{C}(e)$, that is assigned the same type under any expression splicing scene.

Theorem 4.19 (Candidate Expansion Expression Expressibility)**.** *Both of the following hold:*

1. *If* $\Delta\ \Gamma \vdash e : \tau$ *then* $\Delta\ \Gamma \vdash^{\mathbb{E}} \mathcal{C}(e) \rightsquigarrow e : \tau$.
2. *If* $\Delta\ \Gamma \vdash r : \tau \Mapsto \tau'$ *then* $\Delta\ \Gamma \vdash^{\mathbb{E}} \mathcal{C}(r) \rightsquigarrow r : \tau \Mapsto \tau'$.

Proof. By mutual rule induction over Rules (4.1) and Rule (4.2).

For part 1, we induct on the assumption.

Case (4.1a) through (4.1j)**.** In each of these cases, we apply Lemma 3.18 to or over each type formation premise, the IH (part 1) to or over each typing premise, then apply the corresponding ce-expression validation rule in Rules (4.7a) through (4.7j).

Case (4.1k)**.** We have:

(1)	$e = \mathtt{match}[n]\{\tau\}(e'; \{r_i\}_{1 \leq i \leq n})$	by assumption
(2)	$\mathcal{C}(e) = \mathtt{cematch}[n]\{\mathcal{C}(\tau)\}(\mathcal{C}(e'); \{\mathcal{C}(r_i)\}_{1 \leq i \leq n})$	by definition of $\mathcal{C}(e)$
(3)	$\Delta\ \Gamma \vdash e' : \tau'$	by assumption

(4) $\Delta \vdash \tau$ type — by assumption
(5) $\{\Delta\ \Gamma \vdash r_i : \tau' \Rrightarrow \tau\}_{1 \leq i \leq n}$ — by assumption
(6) $\Delta\ \Gamma \vdash^{\mathbb{E}} \mathcal{C}(e') \rightsquigarrow e' : \tau'$ — by IH, part 1 on (3)
(7) $\Delta \vdash^{\mathsf{ts}(\mathbb{E})} \mathcal{C}(\tau) \rightsquigarrow \tau$ type — by Lemma 3.19 on (4)
(8) $\{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathcal{C}(r_i) \rightsquigarrow r_i : \tau' \Rrightarrow \tau\}_{1 \leq i \leq n}$ — by IH, part 2 over (5)
(9) $\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{cematch}[n]\{\mathcal{C}(\tau)\}(\mathcal{C}(e'); \{\mathcal{C}(r_i)\}_{1 \leq i \leq n}) \rightsquigarrow \mathtt{match}[n]\{\tau\}(e'; \{r_i\}_{1 \leq i \leq n}) : \tau$ — by Rule (4.7k) on (6), (7) and (8)

For part 2, we induct on the assumption. There is only one case.

Case (4.2)**.** We have:

(1) $r = \mathtt{rule}(p.e)$ — by assumption
(2) $\mathcal{C}(r) = \mathtt{cerule}(p.\mathcal{C}(e))$ — by definition of $\mathcal{C}(r)$
(3) $\Delta \vdash p : \tau \dashv\!\mid Y$ — by assumption
(4) $\Delta\ \Gamma \cup Y \vdash e : \tau'$ — by assumption
(5) $\Delta\ \Gamma \cup Y \vdash^{\mathbb{E}} \mathcal{C}(e) \rightsquigarrow e : \tau'$ — by IH, part 1 on (4)
(6) $\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{cerule}(p.\mathcal{C}(e)) \rightsquigarrow \mathtt{rule}(p.e) : \tau \Rrightarrow \tau'$ — by Rule (4.8) on (3) and (5)

□

Candidate Expansion Pattern Validation

upTSMs generate candidate expansions of ce-pattern form, as described in Sec. 4.2.7. The *ce-pattern validation judgement*, $\vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : \tau \dashv\!\mid \hat{Y}$, which appears as the final premise of Rule (4.4m), validates ce-patterns by checking that the pattern matches values of type τ, and simultaneously generates the final expansion, p, and the hypotheses $\hat{Y}$. Hypotheses can be generated only by spliced subpatterns, so there is no ce-pattern form corresponding to variable patterns (this is also why $\hat{Y}$ appears as a superscript). The pattern splicing scene, $\mathbb{P}$, is used to "remember" the upTSM context and literal body from the upTSM application site (cf. Rule (4.6f)).

The ce-pattern validation judgement is defined mutually inductively with Rules (4.6) by the following rules.

$$\frac{}{\vdash^{\mathbb{P}} \mathtt{cewildp} \rightsquigarrow \mathtt{wildp} : \tau \dashv\!\mid^{\langle \emptyset ; \emptyset \rangle}} \tag{4.9a}$$

$$\frac{\vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : [\mathtt{rec}(t.\tau)/t]\tau \dashv\!\mid^{\hat{Y}}}{\vdash^{\mathbb{P}} \mathtt{cefoldp}(\grave{p}) \rightsquigarrow \mathtt{foldp}(p) : \mathtt{rec}(t.\tau) \dashv\!\mid^{\hat{Y}}} \tag{4.9b}$$

$$\frac{\{\vdash^{\mathbb{P}} \hat{p}_i \rightsquigarrow p_i : \tau_i \dashv\!\shortmid\!\shortmid^{\hat{\mathcal{Y}}_i}\}_{i \in L}}{\left(\begin{array}{c} \vdash^{\mathbb{P}} \mathtt{cetplp}[L](\{i \hookrightarrow \hat{p}_i\}_{i \in L}) \\ \rightsquigarrow \\ \mathtt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\!\shortmid\!\shortmid^{\cup_{i \in L} \hat{\mathcal{Y}}_i} \end{array}\right)} \tag{4.9c}$$

$$\frac{\vdash^{\mathbb{P}} \hat{p} \rightsquigarrow p : \tau \dashv\!\shortmid\!\shortmid^{\hat{\mathcal{Y}}}}{\vdash^{\mathbb{P}} \mathtt{ceinp}[\ell](\hat{p}) \rightsquigarrow \mathtt{inp}[\ell](p) : \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\!\shortmid\!\shortmid^{\hat{\mathcal{Y}}}} \tag{4.9d}$$

$$\frac{\mathsf{parseUPat}(\mathsf{subseq}(b; m; n)) = \hat{p} \qquad \Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\shortmid\!\shortmid \hat{\mathcal{Y}}}{\vdash^{\Delta; \hat{\Phi}; b} \mathtt{cesplicedp}[m; n] \rightsquigarrow p : \tau \dashv\!\shortmid\!\shortmid^{\hat{\mathcal{Y}}}} \tag{4.9e}$$

Rules (4.9a) through (4.9d) handle ce-patterns of shared form, and correspond to Rules (4.3b) through (4.3e). Rule (4.9e) handles 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 upTSM context Φ from the upTSM application site, producing the hypotheses $\mathcal{Y}$. 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 upTSMs cannot introduce any hypotheses other than those that come from such spliced subpatterns. This achieves the "no hidden assumptions" hygiene property described in Sec. 4.1.5.

The following lemma establishes that every well-typed expanded pattern that generates no hypotheses can be expressed as a ce-pattern.

Lemma 4.20 (Candidate Expansion Pattern Expressibility)**.** *If* $\Delta \vdash p : \tau \dashv\!\shortmid\!\shortmid \emptyset$ *then* $\vdash^{\Delta; \hat{\Phi}; b} \mathcal{C}(p) \rightsquigarrow p : \tau \dashv\!\shortmid\!\shortmid^{\langle \emptyset; \emptyset \rangle}$.

Proof. By rule induction over Rules (4.3).

Case (4.3a)**.** This case does not apply.

Case (4.3b)**.** We have:

(1)	$p = \mathtt{wildp}$	by assumption
(2)	$\mathcal{C}(p) = \mathtt{cewildp}$	by definition of $\mathcal{C}(p)$
(3)	$\vdash^{\Delta; \hat{\Phi}; b} \mathtt{cewildp} \rightsquigarrow \mathtt{wildp} : \tau \dashv\!\shortmid\!\shortmid^{\langle \emptyset; \emptyset \rangle}$	by Rule (4.9a)

Case (4.3c)**.** We have:

(1)	$p = \mathtt{foldp}(p')$	by assumption
(2)	$\mathcal{C}(p) = \mathtt{cefoldp}(\mathcal{C}(p'))$	by definition of $\mathcal{C}(p)$
(3)	$\tau = \mathtt{rec}(t.\tau')$	by assumption
(4)	$\Delta \vdash p' : [\mathtt{rec}(t.\tau')/t]\tau' \dashv\!\shortmid\!\shortmid \emptyset$	by assumption
(5)	$\vdash^{\Delta; \hat{\Phi}; b} \mathcal{C}(p') \rightsquigarrow p : [\mathtt{rec}(t.\tau')/t]\tau' \dashv\!\shortmid\!\shortmid^{\langle \emptyset; \emptyset \rangle}$	by IH on (4)
(6)	$\vdash^{\Delta; \hat{\Phi}; b} \mathtt{cefoldp}(\mathcal{C}(p')) \rightsquigarrow \mathtt{foldp}(p') : \mathtt{rec}(t.\tau') \dashv\!\shortmid\!\shortmid^{\langle \emptyset; \emptyset \rangle}$	by Rule (4.9b) on (5)

Case (4.3d)**.** We have:

(1) $p = \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L})$ by assumption
(2) $\mathcal{C}(p) = \texttt{cetpl}[L](\{i \hookrightarrow \mathcal{C}(p_i)\}_{i \in L})$ by definition of $\mathcal{C}(p)$
(3) $\tau = \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})$ by assumption
(4) $\{\Delta \vdash p_i : \tau_i \dashv\!\mid\! \varnothing\}_{i \in L}$ by assumption
(5) $\{\vdash^{\Delta;\hat{\Phi};b} \mathcal{C}(p_i) \rightsquigarrow p_i : \tau_i \dashv\!\mid\!^{\langle\varnothing;\varnothing\rangle}\}_{i \in L}$ by IH over (4)
(6) $\vdash^{\Delta;\hat{\Phi};b} \texttt{cetpl}[L](\{i \hookrightarrow \mathcal{C}(p_i)\}_{i \in L}) \rightsquigarrow \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\!\mid\!^{\langle\varnothing;\varnothing\rangle}$ by Rule (4.9c) on (5)

Case (4.3e)**.** We have:

(1) $p = \texttt{inp}[\ell](p')$ by assumption
(2) $\mathcal{C}(p) = \texttt{ceinp}[\ell](\mathcal{C}(p'))$ by definition of $\mathcal{C}(p)$
(3) $\tau = \texttt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau')$ by assumption
(4) $\Delta \vdash p' : \tau' \dashv\!\mid\! \varnothing$ by assumption
(5) $\vdash^{\Delta;\hat{\Phi};b} \mathcal{C}(p') \rightsquigarrow p' : \tau' \dashv\!\mid\!^{\langle\varnothing;\varnothing\rangle}$ by IH on (4)
(6) $\vdash^{\Delta;\hat{\Phi};b} \texttt{ceinp}[\ell](\mathcal{C}(p')) \rightsquigarrow \texttt{inp}[\ell](p') : \texttt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau') \dashv\!\mid\!^{\langle\varnothing;\varnothing\rangle}$ by Rule (4.9d) on (5)

□

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 be stated mutually with the corresponding theorem about candidate expansion patterns, because the judgements are mutually defined.

Theorem 4.21 (Typed Pattern Expansion)**.** *Both of the following hold:*

1. *If* $\Delta \vdash_{\langle\mathcal{A};\Phi\rangle} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid\! \langle\mathcal{G};\Upsilon\rangle$ *then* $\Delta \vdash p : \tau \dashv\!\mid\! \Upsilon$.
2. *If* $\vdash^{\Delta;\langle\mathcal{A};\Phi\rangle;b} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid\!^{\langle\mathcal{G};\Upsilon\rangle}$ *then* $\Delta \vdash p : \tau \dashv\!\mid\! \Upsilon$.

Proof. By mutual rule induction on Rules (4.6) and Rules (4.9).

1. We induct on the premise. In the following, let $\hat{\Upsilon} = \langle\mathcal{G};\Upsilon\rangle$ and $\hat{\Phi} = \langle\mathcal{A};\Phi\rangle$.
   Case (4.6a)**.** We have:
   (1) $\hat{p} = \hat{x}$ by assumption
   (2) $p = x$ by assumption
   (3) $\Upsilon = x : \tau$ by assumption
   (4) $\Delta \vdash x : \tau \dashv\!\mid\! x : \tau$ by Rule (4.3a)
   Case (4.6b)**.** We have:
   (1) $p = \texttt{wildp}$ by assumption
   (2) $\Upsilon = \varnothing$ by assumption
   (3) $\Delta \vdash \texttt{wildp} : \tau \dashv\!\mid\! \varnothing$ by Rule (4.3b)

Case (4.6c)**.** We have:

(1) $\hat{p} = \texttt{ufoldp}(\hat{p}')$ by assumption
(2) $p = \texttt{foldp}(p')$ by assumption
(3) $\tau = \texttt{rec}(t.\tau')$ by assumption
(4) $\Delta \vdash_{\hat{\Phi}} \hat{p}' \rightsquigarrow p' : [\texttt{rec}(t.\tau')/t]\tau' \dashv\!\mid \hat{\Upsilon}$ by assumption
(5) $\Delta \vdash p' : [\texttt{rec}(t.\tau')/t]\tau' \dashv\!\mid \Upsilon$ by IH, part 1 on (4)
(6) $\Delta \vdash \texttt{foldp}(p') : \texttt{rec}(t.\tau') \dashv\!\mid \Upsilon$ by Rule (4.3c) on (5)

Case (4.6d)**.** We have:

(1) $\hat{p} = \texttt{utplp}[L](\{i \hookrightarrow \hat{p}_i\}_{i \in L})$ 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) $\{\Delta \vdash_{\hat{\Phi}} \hat{p}_i \rightsquigarrow p_i : \tau_i \dashv\!\mid \langle \mathcal{G}_i; \Upsilon_i \rangle\}_{i \in L}$ by assumption
(5) $\Upsilon = \cup_{i \in L} \Upsilon_i$ by assumption
(6) $\{\Delta \vdash p_i : \tau_i \dashv\!\mid \Upsilon_i\}_{i \in L}$ by IH, part 1 over (4)
(7) $\Delta \vdash \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\!\mid \cup_{i \in L} \Upsilon_i$ by Rule (4.3d) on (6)

Case (4.6e)**.** We have:

(1) $\hat{p} = \texttt{uinp}[\ell](\hat{p}')$ by assumption
(2) $p = \texttt{inp}[\ell](p')$ by assumption
(3) $\tau = \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau')$ by assumption
(4) $\Delta \vdash_{\hat{\Phi}} \hat{p}' \rightsquigarrow p' : \tau' \dashv\!\mid \hat{\Upsilon}$ by assumption
(5) $\Delta \vdash p' : \tau' \dashv\!\mid \Upsilon$ by IH, part 1 on (4)
(6) $\Delta \vdash \texttt{inp}[\ell](p') : \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau') \dashv\!\mid \Upsilon$ by Rule (4.3e) on (5)

Case (4.6f)**.** We have:

(1) $\hat{p} = \texttt{uapuptsm}[b][\hat{a}]$ by assumption
(2) $\mathcal{A} = \mathcal{A}', \hat{a} \rightsquigarrow a$ by assumption
(3) $\Phi = \Phi', a \hookrightarrow \texttt{uptsm}(\tau; e_{\text{parse}})$ by assumption
(4) $b \downarrow e_{\text{body}}$ by assumption
(5) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}\ \texttt{Success} \cdot e_{\text{cand}}$ by assumption
(6) $e_{\text{cand}} \uparrow_{\mathsf{CEPat}} \grave{p}$ by assumption
(7) $\vdash^{\Delta; \langle \mathcal{A}', \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{uptsm}(\tau; e_{\text{parse}}) \rangle; b} \grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\langle \mathcal{G}; \Upsilon \rangle}$ by assumption
(8) $\Delta \vdash p : \tau \dashv\!\mid \Upsilon$ by IH, part 2 on (7)

2. We induct on the premise. In the following, let $\hat{\Upsilon} = \langle \mathcal{G}; \Upsilon \rangle$ and $\hat{\Phi} = \langle \mathcal{A}; \Phi \rangle$.

Case (4.9a)**.** We have:

(1) $p = \texttt{wildp}$ by assumption
(2) $\Upsilon = \emptyset$ by assumption
(3) $\Delta \vdash \texttt{wildp} : \tau \dashv\!\mid \emptyset$ by Rule (4.3b)

Case (4.9b)**.** We have:

(1) $\grave{p} = \texttt{cefoldp}(\grave{p}')$ by assumption

(2) $p = \texttt{foldp}(p')$ by assumption
(3) $\tau = \texttt{rec}(t.\tau')$ by assumption
(4) $\Delta \vdash \Phi$ upTSMs by assumption
(5) $\vdash^{\Delta;\hat{\Phi};b} \grave{p}' \rightsquigarrow p' : [\texttt{rec}(t.\tau')/t]\tau' \dashv\!\mid^{\hat{Y}}$ by assumption
(6) $\Delta \vdash p' : [\texttt{rec}(t.\tau')/t]\tau' \dashv\!\mid Y$ by IH, part 2 on (5) and (4)
(7) $\Delta \vdash \texttt{foldp}(p') : \texttt{rec}(t.\tau') \dashv\!\mid Y$ by Rule (4.3c) on (6)

Case (4.9c)**.** We have:

(1) $\grave{p} = \texttt{cetplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i \in L})$ 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) $\{\vdash^{\Delta;\hat{\Phi};b} \grave{p}_i \rightsquigarrow p_i : \tau_i \dashv\!\mid^{\langle \mathcal{G}_i; Y_i \rangle}\}_{i \in L}$ by assumption
(5) $Y = \cup_{i \in L} Y_i$ by assumption
(6) $\{\Delta \vdash p_i : \tau_i \dashv\!\mid Y_i\}_{i \in L}$ by IH, part 2 over (4)
(7) $\Delta \vdash \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\!\mid \cup_{i \in L} Y_i$ by Rule (4.3d) on (6)

Case (4.9d)**.** We have:

(1) $\grave{p} = \texttt{ceinp}[\ell](\grave{p}')$ by assumption
(2) $p = \texttt{inp}[\ell](p')$ by assumption
(3) $\tau = \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau')$ by assumption
(4) $\vdash^{\Delta;\hat{\Phi};b} \grave{p}' \rightsquigarrow p' : \tau' \dashv\!\mid^{\hat{Y}}$ by assumption
(5) $\Delta \vdash p' : \tau' \dashv\!\mid Y$ by IH, part 2 on (4)
(6) $\Delta \vdash \texttt{inp}[\ell](p') : \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau') \dashv\!\mid Y$ by Rule (4.3e) on (5)

Case (4.9e)**.** We have:

(1) $\grave{p} = \texttt{cesplicedp}[m; n]$ by assumption
(2) $\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{p}$ by assumption
(3) $\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \hat{Y}$ by assumption
(4) $\Delta \vdash p : \tau \dashv\!\mid Y$ 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:

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

where $\|b\|$ is the length of b and $\|\hat{p}\|$ is the sum of the lengths of the literal bodies in $\hat{p}$,

$$\|\hat{x}\| = 0$$
$$\|\texttt{ufoldp}(\hat{p})\| = \|\hat{p}\|$$
$$\|\texttt{utplp}[L](\{i \hookrightarrow \hat{p}_i\}_{i \in L})\| = \sum_{i \in L} \|\hat{p}_i\|$$

$$\|\texttt{uinp[}\ell\texttt{](}\hat{p}\texttt{)}\| = \|\hat{p}\|$$
$$\|\texttt{uapuptsm}[b]\texttt{[}\hat{a}\texttt{]}\| = \|b\|$$

The only case in the proof of part 1 that invokes part 2 is Case (4.6f). There, we have that the metric remains stable:

$$\begin{aligned}
&\|\Delta \vdash_{\hat{\Phi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uetsm}(\tau;e_{\text{parse}})} \texttt{uapuptsm}[b]\texttt{[}\hat{a}\texttt{]} \rightsquigarrow p : \tau \dashv\!\| \hat{Y}\| \\
=&\|\vdash^{\Delta;\hat{\Phi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uetsm}(\tau;e_{\text{parse}});b} \grave{p} \rightsquigarrow p : \tau \dashv\!\|^{\hat{Y}}\| \\
=&\|b\|
\end{aligned}$$

The only case in the proof of part 2 that invokes part 1 is Case (4.9e). There, we have that $\mathsf{parseUPat}(\mathsf{subseq}(b;m;n)) = \hat{p}$ and the IH is applied to the judgement $\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\| \hat{Y}$. 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:

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

i.e. by the definitions above,

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

This is established by appeal to Condition 3.22, 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 4.22 (Pattern Parsing Monotonicity)**.** *If* $\mathsf{parseUPat}(b) = \hat{p}$ *then* $\|\hat{p}\| < \|b\|$.

Combining Conditions 3.22 and 4.22, 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 4.23 (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* $\langle \mathcal{D};\Delta\rangle\ \langle \mathcal{G};\Gamma\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \rightsquigarrow r : \tau \Rrightarrow \tau'$ *then* $\Delta\ \Gamma \vdash r : \tau \Rrightarrow \tau'$.
2. (a) *If* $\Delta\ \Gamma \vdash^{\langle \mathcal{D};\Delta_{app}\rangle;\langle \mathcal{G};\Gamma_{app}\rangle;\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e : \tau$ *and* $\Delta \cap \Delta_{app} = \emptyset$ *and* $dom(\Gamma) \cap dom(\Gamma_{app}) = \emptyset$ *then* $\Delta \cup \Delta_{app}\ \Gamma \cup \Gamma_{app} \vdash e : \tau$.
   (b) *If* $\Delta\ \Gamma \vdash^{\langle \mathcal{D};\Delta_{app}\rangle;\langle \mathcal{G};\Gamma_{app}\rangle;\hat{\Psi};\hat{\Phi};b} \grave{r} \rightsquigarrow r : \tau \Rrightarrow \tau'$ *and* $\Delta \cap \Delta_{app} = \emptyset$ *and* $dom(\Gamma) \cap dom(\Gamma_{app}) = \emptyset$ *then* $\Delta \cup \Delta_{app}\ \Gamma \cup \Gamma_{app} \vdash r : \tau \Rrightarrow \tau'$.

Proof. By mutual rule induction on Rules (4.4), Rule (4.5), Rules (4.7) and Rule (4.8).

1. In the following, let $\hat{\Delta} = \langle \mathcal{D};\Delta\rangle$ and $\hat{\Gamma} = \langle \mathcal{G};\Gamma\rangle$.

(a) We induct on the premise.

Case (4.4a) through (4.4j)**.** These cases follow like the corresponding cases in the proof of Theorem 3.21, i.e. we apply Lemma 3.12 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 (4.1a) through (4.1j).

Case (4.4k)**.** We have:

(1)	$\hat{e} = \texttt{umatch}[n]\{\hat{\tau}\}(\hat{e}'; \{\hat{r}_i\}_{1 \leq i \leq n})$	by assumption
(2)	$e = \texttt{match}[n]\{\tau\}(e'; \{r_i\}_{1 \leq i \leq n})$	by assumption
(3)	$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \rightsquigarrow e' : \tau'$	by assumption
(4)	$\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type}$	by assumption
(5)	$\{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \rightsquigarrow r_i : \tau' \Rightarrow \tau\}_{1 \leq i \leq n}$	by assumption
(6)	$\Delta\ \Gamma \vdash e' : \tau'$	by IH, part 1(a) on (3)
(7)	$\Delta \vdash \tau\ \mathsf{type}$	by Lemma 3.12 on (4)
(8)	$\{\Delta\ \Gamma \vdash r_i : \tau' \Rightarrow \tau\}_{1 \leq i \leq n}$	by IH, part 1(b) over (5)
(9)	$\Delta\ \Gamma \vdash \texttt{match}[n]\{\tau\}(e'; \{r_i\}_{1 \leq i \leq n}) : \tau$	by Rule (4.1k) on (6), (7) and (8)

Case (4.4l)**.** We have:

(1)	$\hat{e} = \texttt{usyntaxue}\{e_{\text{parse}}\}\{\hat{\tau}'\}(\hat{a}.\hat{e}')$	by assumption
(2)	$\hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'\ \mathsf{type}$	by assumption
(3)	$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi},\hat{a} \rightsquigarrow a \hookrightarrow \texttt{uetsm}(\tau'; e_{\text{parse}});\hat{\Phi}} \hat{e}' \rightsquigarrow e : \tau$	by assumption
(4)	$\Delta\ \Gamma \vdash e : \tau$	by IH, part 1(a) on (3)

Case (4.4m)**.** We have:

(1)	$\hat{e} = \texttt{uapuetsm}[b]\,[\hat{a}]$	by assumption
(2)	$\hat{\Psi} = \hat{\Psi}', \hat{a} \rightsquigarrow a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}})$	by assumption
(3)	$b \downarrow e_{\text{body}}$	by assumption
(4)	$e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj Success} \cdot e_{\text{cand}}$	by assumption
(5)	$e_{\text{cand}} \uparrow_{\mathsf{CEExp}} \grave{e}$	by assumption
(6)	$\emptyset\ \emptyset \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e : \tau$	by assumption
(7)	$\emptyset \cap \Delta = \emptyset$	by finite set intersection identity
(8)	$\emptyset \cap \mathrm{dom}(\Gamma) = \emptyset$	by finite set intersection identity
(9)	$\emptyset \cup \Delta\ \emptyset \cup \Gamma \vdash e : \tau$	by IH, part 2(a) on (6), (7), and (8)
(10)	$\Delta\ \Gamma \vdash e : \tau$	by definition of finite set and finite function union over (9)

Case (4.4n)**.** 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) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uptsm}(\tau';e_{\text{parse}})} \hat{e}' \rightsquigarrow e : \tau$ by assumption
(4) $\Delta\ \Gamma \vdash e : \tau$ by IH, part 1(a) on (3)

(b) We induct on the premise. There is only one case.

Case (4.5)**.** We have:

(1) $\hat{r} = \texttt{urule}(\hat{p}.\hat{e})$ by assumption
(2) $r = \texttt{rule}(p.e)$ by assumption
(3) $\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!| \langle \mathcal{A}'; \Upsilon \rangle$ by assumption
(4) $\hat{\Delta}\ \langle \mathcal{A} \uplus \mathcal{A}'; \Gamma \cup \Upsilon \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau'$ by assumption
(5) $\Delta \vdash p : \tau \dashv\!| \Upsilon$ by Theorem 4.21, part 1 on (3)
(6) $\Delta\ \Gamma \cup \Upsilon \vdash e : \tau'$ by IH, part 1(a) on (4)
(7) $\Delta\ \Gamma \vdash \texttt{rule}(p.e) : \tau \Rightarrow \tau'$ by Rule (4.2) 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) We induct on the premise.

Case (4.7a) through (4.7j)**.** These cases follow like the analagous cases in the proof of Theorem 3.21, i.e. we apply the IH, part 2(a) to all ce-expression validation judgements, Lemma 3.19 to all ce-type validation judgements, the identification convention to ensure that extended contexts remain disjoint, weakening and exchange as needed, and conclude by applying the corresponding typing rule in Rules (4.1a) through (4.1j).

Case (4.7k)**.** We have:

(1) $\grave{e} = \texttt{cematch}[n]\{\grave{\tau}\}(\grave{e}'; \{\grave{r}_i\}_{1 \leq i \leq n})$ 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 3.19 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 (4.1k) on (8), (9), (10)

Case (4.7l)**.** We have:

(1) $\grave{e} = \texttt{cesplicede}[m;n]$ by assumption

(2) $\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e}$ — by assumption
(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau$ — 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 4.1 over Δ and Γ and exchange on (6)

(b) We induct on the premise. There is only one case.

Case (4.8)**.** We have:

(1) $\grave{r} = \texttt{cerule}(p.\grave{e})$ — by assumption
(2) $r = \texttt{rule}(p.e)$ — by assumption
(3) $\Delta \vdash p : \tau \dashv\mid \Upsilon$ — by assumption
(4) $\Delta\ \Gamma \cup \Upsilon \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e : \tau'$ — by assumption
(5) $\Delta \cap \Delta_{\text{app}} = \emptyset$ — by assumption
(6) $\text{dom}(\Gamma) \cap \text{dom}(\Upsilon) = \emptyset$ — by identification convention
(7) $\text{dom}(\Gamma_{\text{app}}) \cap \text{dom}(\Upsilon) = \emptyset$ — by identification convention
(8) $\text{dom}(\Gamma) \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset$ — by assumption
(9) $\text{dom}(\Gamma \cup \Upsilon) \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset$ — by standard finite set definitions and identities on (6), (7) and (8)
(10) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Upsilon \cup \Gamma_{\text{app}} \vdash e : \tau'$ — by IH, part 2(a) on (4), (5) and (9)
(11) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \cup \Upsilon \vdash e : \tau'$ — by exchange of Υ 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 (4.2) 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 ueTSM literal bodies

in $\hat{e}$,

$$\begin{aligned}
\|\hat{x}\| &= 0 \\
\|\texttt{ulam\{}\hat{\tau}\texttt{\}(}\hat{x}.\hat{e}\texttt{)}\| &= \|\hat{e}\| \\
\|\texttt{uap(}\hat{e}_1\texttt{;}\hat{e}_2\texttt{)}\| &= \|\hat{e}_1\| + \|\hat{e}_2\| \\
\|\texttt{utlam(}\hat{t}.\hat{e}\texttt{)}\| &= \|\hat{e}\| \\
\|\texttt{utap\{}\hat{\tau}\texttt{\}(}\hat{e}\texttt{)}\| &= \|\hat{e}\| \\
\|\texttt{ufold\{}\hat{t}.\hat{\tau}\texttt{\}(}\hat{e}\texttt{)}\| &= \|\hat{e}\| \\
\|\texttt{uunfold(}\hat{e}\texttt{)}\| &= \|\hat{e}\| \\
\|\texttt{utpl}[L]\texttt{(}\{i \hookrightarrow \hat{e}_i\}_{i\in L}\texttt{)}\| &= \sum_{i\in L} \|\hat{e}_i\| \\
\|\texttt{upr}[\ell]\texttt{(}\hat{e}\texttt{)}\| &= \|\hat{e}\| \\
\|\texttt{uin}[L][\ell]\texttt{\{}\{i \hookrightarrow \hat{\tau}_i\}_{i\in L}\texttt{\}(}\hat{e}\texttt{)}\| &= \|\hat{e}\| \\
\|\texttt{umatch}[n]\texttt{\{}\hat{\tau}\texttt{\}(}\hat{e}\texttt{;}\{\hat{r}_i\}_{1\le i\le n}\texttt{)}\| &= \|\hat{e}\| + \sum_{1\le i\le n} \|r_i\| \\
\|\texttt{usyntaxue\{}e_{\text{parse}}\texttt{\}\{}\hat{\tau}\texttt{\}(}\hat{a}.\hat{e}\texttt{)}\| &= \|\hat{e}\| \\
\|\texttt{uapuetsm}[b]\texttt{[}\hat{a}\texttt{]}\| &= \|b\| \\
\|\texttt{usyntaxup\{}e_{\text{parse}}\texttt{\}\{}\hat{\tau}\texttt{\}(}\hat{a}.\hat{e}\texttt{)}\| &= \|\hat{e}\|
\end{aligned}$$

and $\|r\|$ is defined as follows:

$$\|\texttt{urule(}\hat{p}.\hat{e}\texttt{)}\| = \|\hat{e}\|$$

The only case in the proof of part 1 that invokes part 2 is Case (4.4m). There, we have that the metric remains stable:

$$\begin{aligned}
&\|\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uetsm(}\tau\texttt{;}e_{\text{parse}}\texttt{);}\hat{\Phi}} \texttt{uapuetsm}[b]\texttt{[}\hat{a}\texttt{]} \rightsquigarrow e : \tau\| \\
=&\|\varnothing\ \varnothing \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uetsm(}\tau\texttt{;}e_{\text{parse}}\texttt{);}\hat{\Phi};b} \grave{e} \rightsquigarrow e : \tau\| \\
=&\|b\|
\end{aligned}$$

The only case in the proof of part 2 that invokes part 1 is Case (4.7l). There, we have that $\mathsf{parseUExp}(\mathsf{subseq}(b;m;n)) = \hat{e}$ and the IH is applied to the judgement $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau$. Because the metric is stable when passing from part 1 to part 2, we must have that it is strictly decreasing in the other direction:

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

i.e. by the definitions above,

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

This is established by appeal to Condition 3.22, 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 apply a TSM and delimit each literal body.

Condition 4.24 (Expression Parsing Monotonicity). *If* $\mathsf{parseUExp}(b) = \hat{e}$ *then* $\|\hat{e}\| < \|b\|$.

Combining Conditions 3.22 and 4.24, we have that $\|\hat{e}\| < \|b\|$ as needed. □

Part II

Parametric TSMs

Chapter 5

Parameterized TSMs (pTSMs)

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

— Bob Ross, *The Joy of Painting*

In the preceding chapters, we introduced unparameterized TSMs (uTSMs). uTSMs are defined at a single type, like `Rx`, and the expansions that they generate have access to bindings at the application site only via spliced subterms. In this chapter, we introduce *parameterized TSMs* (pTSMs) – TSMs parameterized by types and modules. pTSMs can be defined over a parameterized family of types, and can access their parameters within the expansions that they generate.

This chapter is organized much 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 formalize parameterized TSMs by defining a reduced calculus, miniVerse$_\forall$, in Sec. 5.2.

5.1 Parameterized TSMs By Example

5.1.1 Type Parameters

VerseML, like many ML dialects, provides support for defining *type constructors*, which express type-parameterized families of types. The canonical example is the type constructor `list`, which constructs list types given one type parameter, the *element type*. In VerseML, `list` is defined as follows.

```
type list('a) = Nil | Cons of 'a * list('a)
```

For any type `T`, the type of lists containing elements of type `T` can be constructed by applying `list` to `T`, written `list(T)`.[1] In other words, type constructors can be understood as total functions at the level of *constructor expressions*.

[1] As with function application, the parentheses are optional (though for type constructor application, it is typical to include them). In many other ML dialects, type parameters are given in prefix form, e.g. in Standard ML, one writes `int list` rather than `list(int)`.

Kinds classify constructor expressions, much like types classify expressions. Types are constructor expressions of kind `T`, and type constructors are constructor expressions of arrow kind. Here, `list` takes a single type parameter, so it has arrow kind `T -> T`.

As discussed in Sec. **??**, full-scale ML dialects commonly define derived syntactic forms that decrease the syntactic cost of introducing and pattern matching over values of list type. VerseML, in contrast, does not build in such derived list forms.

In lieu of derived forms for introducing lists, we define the following *parameterized expression TSM* (peTSM):

```
1 syntax $list('a) at list('a) {
2   static fn(body : Body) : ParseResult(CEExp) => (* ... *)
3 }
```

Line 1 names the peTSM `$list` and specifies a single type parameter, `'a` (implicitly of kind `T`). This type parameter appears in the type annotation, which specifies that `$list`, when applied to a type `'a` and a generalized literal form, will only generate expansions of type `list('a)`. For example, we can apply `$list` to the type `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]
```

Line 2 of the definition of `$list` defines its parse function. Parse functions operate as described in Chapter 3 to generate encodings of candidate expansions, which are subsequently validated to generate the final expansions of expressions of peTSM application form, like those in the example above. For `$list`, the parse function (whose body is elided above for concision) breaks the literal body up into spliced subexpressions – those separated by commas become individual elements at the head of the list being generated, and, optionally, a trailing spliced subexpression prefixed by two colons (`::`) becomes the tail of the list being generated (the tail is `Nil` otherwise). The final expansion of the example above, if written textually, is:

```
val y : list(int) = Cons(3, Cons(4, Cons(5, Nil)))
val x : list(int) = Cons(1, Cons(2, y))
```

Similarly, in lieu of derived list pattern forms, we define the following *parameterized pattern TSM* (ppTSM):

```
syntax $list('a) at list('a) for patterns {
  static fn(body : Body) : ParseResult(CEPat) => (* ... *)
}
```

Again, Line 1 names the ppTSM `$list` and specifies a single type parameter, `'a`. This type parameter appears in the type annotation, which specifies that `$list`, when apply to a type `'a` and a generalized literal form, will only generate patterns that match values of type `list('a)`.

For example, we can apply the ppTSM `$list` and the `$list` to define the polymorphic map function as follows.

```
fun map (f : 'a -> 'b) (x : list('a)) => match x {
  $list('a) [] => $list('b) []
| $list('a) [hd :: tl] => $list('b) [f hd :: map f tl]
```

```
}
```

The expansion of this function definition, written textually, is:

```
fun map (f : 'a -> 'b) (x : list('a)) : 'b list => match x {
  Nil => Nil
| Cons(hd, tl) => Cons(f hd, map f tl)
}
```

This is somewhat unsatisfying, however, because the expansion is more concise than the unexpanded definition of `map`. To further reduce syntactic cost, we can designate `$list` as the implicit TSM for both expressions and patterns at all types `'a list` around our definition of `map` as follows.

```
implicit syntax $list('a) in
  fun map (f : 'a -> 'b) (x : 'a list) : 'b list => match x {
    [] => []
  | [hd :: tl] => [f hd :: map f tl]
  }
end
```

By designating an implicit TSM, we no longer need to explicitly apply `$list` within expressions in analytic position or patterns.

5.1.2 Module Parameters

VerseML also provides a module language based on the Standard ML module language [52]. The module language consists of *module expressions* classified by *signatures*.

The canonical example is the signature for working with persistent queues.

```
signature QUEUE = sig
  type queue('a)
  val empty  : queue('a)
  val insert : 'a * queue('a) -> queue('a)
  val remove : queue('a) -> option('a * queue('a))
end
```

Structures that match this signature must define a type constructor `queue` of kind `T -> T` and three values – `empty` introduces the empty queue, `insert` inserts a value onto the back of a queue, and `remove` removes the element at the front of the queue and returns it and the remaining queue, or `None` if the queue is empty.

There are many possible structures that implement this signature. For example, we can define a structure `ListQueue` that represents queues internally as lists, where the head of the list is the back of the queue. With this representation, `insert` is a constant time operation, but `remove` is a linear time operation. Alternatively, we might define a structure `TwoListQueue` that represents queues internally as a pair of lists, maintaining the invariant that one is the reverse of the other, so that both `insert` and `remove` are constant time operations (see [36] for the details of this and other possibilities).

Regardless of the implementation that the client chooses, we would like for the client to be able to introduce queues more naturally and at lower syntactic cost than is possible by directly applying the functions specified by the signature above. In VerseML, we

can give clients of structures matching the signature `QUEUE` this ability by defining the following parameterized expression TSM:

```
syntax $queue(Q : QUEUE)('a) at Q.queue('a) {
  static fn(body : Body) : ParseResult(CEExp) => (* ... *)
}
```

This peTSM specifies one module parameter, `Q`, which must match the signature `QUEUE`, and one type parameter, `'a` (implicitly of kind `T`). These appear in the type annotation, which specifies that expansions that arise from applying `$queue` to a module `Q : QUEUE` and a type `'a` will be of type `Q.queue('a)`. For example:

```
1 val q = $queue TwoListQueue int [> 1, 2, 3]
2 val q' = $queue TwoListQueue int [q > 4, 5]
```

On Line 1, the initial angle bracket (`>`) indicates that the items are inserted in left-to-right order. The items in the queue are given as spliced subexpressions separated by commas. Line 2 inserts two additional items onto the back of the queue `q`. The expansion of this example, written textually, is:

```
1 val q : TwoListQueue.queue(int) =
2   TwoListQueue.insert(1,
3     TwoListQueue.insert(2,
4       TwoListQueue.insert(3,
5         TwoListQueue.empty)))
6 val q' : TwoListQueue.queue(int) =
7   TwoListQueue.insert(4, TwoListQueue.insert(5, q))
```

Notice that the expansion can refer to the module parameter `TwoListQueue`.

We can further reduce syntactic cost by defining a synonym for the partial application of `$queue` to the module parameter `TwoListQueue`:

```
syntax $tlq = $queue TwoListQueue
val q = $tlq int [> 1, 2, 3]
```

We can further define a synonym for the partial application of `$tlq` to a type parameter:

```
syntax $tlqi = $tlq int (* = $queue TwoListQueue int *)
val q' = $tlqi [q > 4, 5]
```

Another way to reduce syntactic cost is by designating `$queue Q 'a` the implicit TSM at all types of the form `Q.queue('a)` where `Q : QUEUE`. This is written as follows:

```
implicit syntax (Q : QUEUE) ('a) => $queue Q 'a in
  val q : TwoListQueue.queue(int) = [> 1, 2, 3]
  val q' : TwoListQueue.queue(int) = [q > 4, 5]
end
```

This designation is particularly useful for clients who need to construct a queue as an argument to a function. For example, consider a function

```
enqueue_jobs : Q.queue(Job) -> Ticket
```

for some module `Q : QUEUE` and types `Job` and `Ticket`. We can enqueue a sequence of jobs `j1` through `j4` under the TSM designation above as follows:

```
enqueue_jobs [> j1, j2, j3, j4]
```

Sort			Operational Form	Stylized Form	Description
Kind	κ	$::=$	$\texttt{darr}(\kappa; u.\kappa)$	$(u :: \kappa) \to \kappa$	dependent function
			$\texttt{unit}$	$\langle\!\rangle$	nullary product
			$\texttt{dprod}(\kappa; u.\kappa)$	$(u :: \kappa) \times \kappa$	dependent product
			$\texttt{Type}$	$\texttt{T}$	types
			$\texttt{S}(\tau)$	$[=\tau]$	singleton
Con	c, τ	$::=$	u	u	variable
			t	t	variable
			$\texttt{abs}(u.c)$	$\lambda u.c$	abstraction
			$\texttt{app}(c; c)$	$c(c)$	application
			$\texttt{triv}$	$\langle\!\rangle$	trivial
			$\texttt{pair}(c; c)$	$\langle\!\langle c, c \rangle\!\rangle$	pair
			$\texttt{prl}(c)$	$c \cdot \texttt{l}$	left projection
			$\texttt{prr}(c)$	$c \cdot \texttt{r}$	right projection
			$\texttt{parr}(\tau; \tau)$	$\tau \rightharpoonup \tau$	partial function
			$\texttt{all}\{\kappa\}(u.\tau)$	$\forall(u :: \kappa).\tau$	polymorphic
			$\texttt{rec}(t.\tau)$	$\mu t.\tau$	recursive
			$\texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})$	$\langle \{i \hookrightarrow \tau_i\}_{i \in L} \rangle$	labeled product
			$\texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L})$	$[\{i \hookrightarrow \tau_i\}_{i \in L}]$	labeled sum
			$\texttt{con}(M)$	$M \cdot \texttt{c}$	constructor part

Figure 5.1: Syntax of kinds and constructors in miniVerse$_\forall$. 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. Kinds and constructors are identified up to α-equivalence.

Sort			Operational Form	Stylized Form	Description
Exp	e	::=	x	x	variable
			$\texttt{lam}\{\tau\}(x.e)$	$\lambda x{:}\tau.e$	abstraction
			$\texttt{ap}(e;e)$	$e(e)$	application
			$\texttt{clam}\{\kappa\}(u.e)$	$\Lambda u{::}\kappa.e$	constructor abstraction
			$\texttt{cap}\{\kappa\}(e)$	$e[\kappa]$	constructor application
			$\texttt{fold}\{t.\tau\}(e)$	$\texttt{fold}(e)$	fold
			$\texttt{unfold}(e)$	$\texttt{unfold}(e)$	unfold
			$\texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L})$	$\langle\{i \hookrightarrow e_i\}_{i\in L}\rangle$	labeled tuple
			$\texttt{pr}[\ell](e)$	$e \cdot \ell$	projection
			$\texttt{in}[L][\ell]\{\{i \hookrightarrow \tau_i\}_{i\in L}\}(e)$	$\texttt{inj}\ \ell \cdot e$	injection
			$\texttt{match}[n]\{\tau\}(e;\{r_i\}_{1\le i\le n})$	$\texttt{match}\ e\ \{r_i\}_{1\le i\le n}$	match
			$\texttt{val}(M)$	$M \cdot \texttt{v}$	value part
Rule	r	::=	$\texttt{rule}(p.e)$	$p \Rightarrow e$	rule
Pat	p	::=	x	x	variable pattern
			$\texttt{wildp}$	$\_$	wildcard pattern
			$\texttt{foldp}(p)$	$\texttt{fold}(p)$	fold pattern
			$\texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i\in L})$	$\langle\{i \hookrightarrow p_i\}_{i\in L}\rangle$	labeled tuple pattern
			$\texttt{inp}[\ell](p)$	$\texttt{inj}\ \ell \cdot p$	injection pattern

Figure 5.2: Syntax of expanded expressions, rules and patterns (collectively, expanded terms) in miniVerse$_\forall$. Expanded terms are identified up to α-equivalence.

Sort			Operational Form	Stylized Form	Description
Sig	σ	::=	$\texttt{sig}\{\kappa\}(u.\tau)$	$[\![u :: \kappa; \tau]\!]$	signature
Mod	M	::=	X	X	variable
			$\texttt{struct}(c;e)$	$[\![c;e]\!]$	structure
			$\texttt{seal}\{\sigma\}(M)$	$M \upharpoonright \sigma$	seal
			$\texttt{mlet}\{\sigma\}(M;X.M)$	$(\texttt{let}\ X = M\ \texttt{in}\ M) : \sigma$	definition

Figure 5.3: Syntax of signatures and module expressions in miniVerse$_\forall$. Signatures and module expressions are identified up to α-equivalence.

5.2 miniVerse$_\forall$

5.2.1 Syntax of the Inner Language

Syntax of the Inner Core Language

We adopt the metatheoretic conventions established for our definitions of miniVerse$_{\mathbf{UE}}$ in Sec. 3.2 and miniVerse$_{\mathbf{U}}$ in Sec. 4.2 without restating them.

Kinds and constructors in Figure 5.1.

Expanded expressions, rules and patterns in Figure 5.2.

Syntax of the Inner Module Language

Module expressions and signatures in Figure 5.3.

5.2.2 Statics of the Inner Language

The *statics of the inner 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 inner constructor language*, define the statics of expanded kinds and constructors.

Judgement Form	**Description**
$\Omega \vdash \kappa$ kind	κ is a well-formed 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 second group of judgements, which we refer to as the *statics of the inner core 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
$\Omega \vdash r : \tau \Rightarrow \tau'$	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'

The third group of judgements, which we refer to as the *statics of the inner module language*, define the statics of expanded signatures and module expressions.

Judgement Form	Description
$\Omega \vdash \sigma$ sig	σ is a well-formed 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 has signature σ
$\Omega \vdash M$ mval	M is a module value

A *unified inner context*, Ω, is a finite function that maps

- each expression variable $x \in \mathrm{dom}(\Omega)$ to the hypothesis $x : \tau$ for some τ;
- each constructor variable $u \in \mathrm{dom}(\Omega)$ to the hypothesis $u :: \kappa$ for some κ; and
- each module variable $X \in \mathrm{dom}(\Omega)$ to the hypothesis $X :: \sigma$ for some σ.

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

A well-formed unified inner context is one that can be constructed by some sequence of such extensions, start from the empty context, $\emptyset$.

Kinds and Constructors

Kind formation TODO: describe these

$$\frac{\Omega \vdash \kappa_1 \text{ kind} \qquad \Omega, u :: \kappa_1 \vdash \kappa_2 \text{ kind}}{\Omega \vdash \mathtt{darr}(\kappa_1; u.\kappa_2) \text{ kind}} \tag{5.1a}$$

$$\frac{}{\Omega \vdash \mathtt{unit} \text{ kind}} \tag{5.1b}$$

$$\frac{\Omega \vdash \kappa_1 \text{ kind} \qquad \Omega, u :: \kappa_1 \vdash \kappa_2 \text{ kind}}{\Omega \vdash \mathtt{dprod}(\kappa_1; u.\kappa_2) \text{ kind}} \tag{5.1c}$$

$$\frac{}{\Omega \vdash \mathtt{Type} \text{ kind}} \tag{5.1d}$$

$$\frac{\Omega \vdash \tau :: \mathtt{Type}}{\Omega \vdash \mathtt{S}(\tau) \text{ kind}} \tag{5.1e}$$

Kind definitional equality TODO: describe these

$$\frac{\Omega \vdash \kappa \text{ kind}}{\Omega \vdash \kappa \equiv \kappa} \tag{5.2a}$$

$$\frac{\Omega \vdash \kappa \equiv \kappa'}{\Omega \vdash \kappa' \equiv \kappa} \tag{5.2b}$$

$$\frac{\Omega \vdash \kappa \equiv \kappa' \qquad \Omega \vdash \kappa' \equiv \kappa''}{\Omega \vdash \kappa \equiv \kappa''} \tag{5.2c}$$

$$\frac{\Omega \vdash \kappa_1 \equiv \kappa_1' \qquad \Omega, u :: \kappa_1 \vdash \kappa_2 \equiv \kappa_2'}{\Omega \vdash \mathtt{darr}(\kappa_1; u.\kappa_2) \equiv \mathtt{darr}(\kappa_1'; u.\kappa_2')} \tag{5.2d}$$

$$\frac{\Omega \vdash \kappa_1 \equiv \kappa_1' \qquad \Omega, u :: \kappa_1 \vdash \kappa_2 \equiv \kappa_2'}{\Omega \vdash \mathtt{dprod}(\kappa_1; u.\kappa_2) \equiv \mathtt{dprod}(\kappa_1'; u.\kappa_2')} \tag{5.2e}$$

$$\frac{\Omega \vdash c \equiv c' :: \mathtt{Type}}{\Omega \vdash \mathtt{S}(c) \equiv \mathtt{S}(c')} \tag{5.2f}$$

Subkinding TODO: describe these

$$\frac{\Omega \vdash \kappa \equiv \kappa'}{\Omega \vdash \kappa <:: \kappa'} \tag{5.3a}$$

$$\frac{\Omega \vdash \kappa <:: \kappa' \qquad \Omega \vdash \kappa' <:: \kappa''}{\Omega \vdash \kappa <:: \kappa''} \tag{5.3b}$$

$$\frac{\Omega \vdash \kappa_1' <:: \kappa_1 \qquad \Omega, u :: \kappa_1' \vdash \kappa_2 <:: \kappa_2'}{\Omega \vdash \mathtt{darr}(\kappa_1; u.\kappa_2) <:: \mathtt{darr}(\kappa_1'; u.\kappa_2')} \tag{5.3c}$$

$$\frac{\Omega \vdash \kappa_1 <:: \kappa_1' \qquad \Omega, u :: \kappa_1 \vdash \kappa_2 <:: \kappa_2'}{\Omega \vdash \mathtt{dprod}(\kappa_1; u.\kappa_2) <:: \mathtt{dprod}(\kappa_1'; u.\kappa_2')} \tag{5.3d}$$

$$\frac{\Omega \vdash \tau :: \mathtt{Type}}{\Omega \vdash \mathtt{S}(\tau) <:: \mathtt{Type}} \tag{5.3e}$$

$$\frac{\Omega \vdash \tau <: \tau'}{\Omega \vdash \mathtt{S}(\tau) <:: \mathtt{S}(\tau')\mathtt{Type}} \tag{5.3f}$$

Kinding TODO: describe these

$$\frac{\Omega \vdash c :: \kappa_1 \qquad \Omega \vdash \kappa_1 <:: \kappa_2}{\Omega \vdash c :: \kappa_2} \tag{5.4a}$$

$$\frac{}{\Omega, u :: \kappa \vdash u :: \kappa} \tag{5.4b}$$

$$\frac{\Omega, u :: \kappa_1 \vdash c_2 :: \kappa_2}{\Omega \vdash \mathtt{abs}(u.c_2) :: \mathtt{darr}(\kappa_1; u.\kappa_2)} \tag{5.4c}$$

$$\frac{\Omega \vdash c_1 :: \mathtt{darr}(\kappa_2; u.\kappa) \qquad \Omega \vdash c_2 :: \kappa_2}{\Omega \vdash \mathtt{app}(c_1; c_2) :: [c_1/u]\kappa} \tag{5.4d}$$

$$\frac{}{\Omega \vdash \mathtt{triv} :: \mathtt{unit}} \tag{5.4e}$$

$$\frac{\Omega \vdash c_1 :: \kappa_1 \qquad \Omega \vdash c_2 :: [c_1/u]\kappa_2}{\Omega \vdash \mathtt{pair}(c_1; c_2) :: \mathtt{dprod}(\kappa_1; u.\kappa_2)} \tag{5.4f}$$

$$\frac{\Omega \vdash c :: \mathtt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash \mathtt{prl}(c) :: \kappa_1} \tag{5.4g}$$

$$\frac{\Omega \vdash c :: \mathtt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash \mathtt{prr}(c) :: [\mathtt{prl}(c)/u]\kappa_2} \tag{5.4h}$$

$$\frac{\Omega \vdash \tau_1 :: \mathtt{Type} \qquad \Omega \vdash \tau_2 :: \mathtt{Type}}{\Omega \vdash \mathtt{parr}(\tau_1; \tau_2) :: \mathtt{Type}} \tag{5.4i}$$

$$\frac{\Omega \vdash \kappa \ \mathsf{kind} \qquad \Omega, u :: \kappa \vdash \tau :: \mathtt{Type}}{\Omega \vdash \mathtt{all}\{\kappa\}(u.\tau) :: \mathtt{Type}} \tag{5.4j}$$

$$\frac{\Omega, t :: \mathtt{Type} \vdash \tau :: \mathtt{Type}}{\Omega \vdash \mathtt{rec}(t.\tau) :: \mathtt{Type}} \tag{5.4k}$$

$$\frac{\{\Omega \vdash \tau_i :: \mathtt{Type}\}_{1 \leq i \leq n}}{\Omega \vdash \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) :: \mathtt{Type}} \tag{5.4l}$$

$$\frac{\{\Omega \vdash \tau_i :: \mathtt{Type}\}_{1 \leq i \leq n}}{\Omega \vdash \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) :: \mathtt{Type}} \tag{5.4m}$$

$$\frac{\Omega \vdash c :: \mathtt{Type}}{\Omega \vdash c :: \mathtt{S}(c)} \tag{5.4n}$$

$$\frac{\Omega \vdash M \ \mathsf{mval} \qquad \Omega \vdash M : \mathtt{sig}\{\kappa\}(u.\tau)}{\Omega \vdash \mathtt{con}(M) :: \kappa} \tag{5.4o}$$

Constructor equality TODO: describe this

$$\frac{\Omega \vdash c :: \kappa}{\Omega \vdash c \equiv c :: \kappa} \tag{5.5a}$$

$$\frac{\Omega \vdash c \equiv c' :: \kappa}{\Omega \vdash c' \equiv c :: \kappa} \tag{5.5b}$$

$$\frac{\Omega \vdash c \equiv c' :: \kappa \qquad \Omega \vdash c' \equiv c'' :: \kappa}{\Omega \vdash c \equiv c'' :: \kappa} \tag{5.5c}$$

$$\frac{\Omega, u :: \kappa_1 \vdash c \equiv c' :: \kappa_2}{\Omega \vdash \mathtt{abs}(u.c) \equiv \mathtt{abs}(u.c') :: \mathtt{darr}(\kappa_1; u.\kappa_2)} \tag{5.5d}$$

$$\frac{\Omega \vdash c_1 \equiv c_1' :: \mathtt{darr}(\kappa_2; u.\kappa) \qquad \Omega \vdash c_2 \equiv c_2' :: \kappa_2}{\Omega \vdash \mathtt{app}(c_1; c_2) \equiv \mathtt{app}(c_1'; c_2') :: \kappa} \tag{5.5e}$$

$$\frac{\Omega \vdash \mathtt{abs}(u.c) :: \mathtt{darr}(\kappa_2; u.\kappa) \qquad \Omega \vdash c_2 :: \kappa_2}{\Omega \vdash \mathtt{app}(\mathtt{abs}(u.c); c_2) \equiv [c_2/u]c :: [c_2/u]\kappa} \tag{5.5f}$$

$$\frac{\Omega \vdash c_1 \equiv c_1' :: \kappa_1 \qquad \Omega \vdash c_2 \equiv c_2' :: [c_1/u]\kappa_2}{\Omega \vdash \mathtt{pair}(c_1; c_2) \equiv \mathtt{pair}(c_1'; c_2') :: \mathtt{dprod}(\kappa_1; u.\kappa_2)} \tag{5.5g}$$

$$\frac{\Omega \vdash c \equiv c' :: \mathtt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash \mathtt{prl}(c) \equiv \mathtt{prl}(c') :: \kappa_1} \tag{5.5h}$$

$$\frac{\Omega \vdash c_1 :: \kappa_1 \qquad \Omega \vdash c_2 :: \kappa_2}{\Omega \vdash \mathtt{prl}(\mathtt{pair}(c_1; c_2)) \equiv c_1 :: \kappa_1} \tag{5.5i}$$

$$\frac{\Omega \vdash c \equiv c' :: \mathtt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash \mathtt{prr}(c) \equiv \mathtt{prr}(c') :: [\mathtt{prl}(c)/u]\kappa_2} \tag{5.5j}$$

$$\frac{\Omega \vdash c_1 :: \kappa_1 \qquad \Omega \vdash c_2 :: \kappa_2}{\Omega \vdash \mathtt{prr}(\mathtt{pair}(c_1; c_2)) \equiv c_2 :: \kappa_2} \tag{5.5k}$$

$$\frac{\Omega \vdash \tau_1 \equiv \tau_1' :: \mathtt{Type} \qquad \Omega \vdash \tau_2 \equiv \tau_2' :: \mathtt{Type}}{\Omega \vdash \mathtt{parr}(\tau_1; \tau_2) \equiv \mathtt{parr}(\tau_1'; \tau_2') :: \mathtt{Type}} \tag{5.5l}$$

$$\frac{\Omega \vdash \kappa \equiv \kappa' \qquad \Omega, u :: \kappa \vdash \tau \equiv \tau' :: \mathtt{Type}}{\Omega \vdash \mathtt{all}\{\kappa\}(u.\tau) \equiv \mathtt{all}\{\kappa'\}(u.\tau') :: \mathtt{Type}} \tag{5.5m}$$

$$\frac{\Omega, t :: \mathtt{Type} \vdash \tau \equiv \tau' :: \mathtt{Type}}{\Omega \vdash \mathtt{rec}(t.\tau) \equiv \mathtt{rec}(t.\tau') :: \mathtt{Type}} \tag{5.5n}$$

$$\frac{\{\Omega \vdash \tau_i \equiv \tau_i' :: \mathtt{Type}\}_{1 \le i \le n}}{\Omega \vdash \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \equiv \mathtt{prod}[L](\{i \hookrightarrow \tau_i'\}_{i \in L}) :: \mathtt{Type}} \tag{5.5o}$$

$$\frac{\{\Omega \vdash \tau_i \equiv \tau_i' :: \mathtt{Type}\}_{1 \le i \le n}}{\Omega \vdash \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \equiv \mathtt{sum}[L](\{i \hookrightarrow \tau_i'\}_{i \in L}) :: \mathtt{Type}} \tag{5.5p}$$

$$\frac{\Omega \vdash c :: \mathtt{S}(c')}{\Omega \vdash c \equiv c' :: \mathtt{Type}} \tag{5.5q}$$

$$\frac{\Omega \vdash \mathtt{struct}(c; e)\ \mathsf{mval} \qquad \Omega \vdash \mathtt{struct}(c; e) : \mathtt{sig}\{\kappa\}(u.\tau)}{\Omega \vdash \mathtt{con}(\mathtt{struct}(c; e)) \equiv c :: \kappa} \tag{5.5r}$$

Expanded Expressions, Rules and Patterns

Types, τ, classify expressions. The constructors of kind Type coincide with the types of miniVerse$_\forall$.

$$\frac{\Omega \vdash \tau :: \mathtt{Type}}{\Omega \vdash \tau\ \mathsf{type}} \tag{5.6}$$

Type equality then coincides with constructor equality at kind $\mathtt{Type}$.

$$\frac{\Omega \vdash \tau \equiv \tau :: \mathtt{Type}}{\Omega \vdash \tau \equiv \tau' \ \mathsf{type}} \tag{5.7}$$

Subtyping.

$$\frac{\Omega \vdash \tau_1 \equiv \tau_2 \ \mathsf{type}}{\Omega \vdash \tau_1 <: \tau_2} \tag{5.8a}$$

$$\frac{\Omega \vdash \tau <: \tau' \qquad \Omega \vdash \tau' <: \tau''}{\Omega \vdash \tau <: \tau''} \tag{5.8b}$$

$$\frac{\Omega \vdash \tau_1' <: \tau_1 \qquad \Omega \vdash \tau_2 <: \tau_2'}{\Omega \vdash \mathtt{parr}(\tau_1; \tau_2) <: \mathtt{parr}(\tau_1'; \tau_2')} \tag{5.8c}$$

$$\frac{\Omega \vdash \kappa <:: \kappa' \qquad \Omega, u :: \kappa \vdash \tau <: \tau'}{\Omega \vdash \mathtt{all}\{\kappa\}(u.\tau) <: \mathtt{all}\{\kappa'\}(u.\tau')} \tag{5.8d}$$

$$\frac{\{\Omega \vdash \tau_i <: \tau_i'\}_{i \in L}}{\Omega \vdash \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) <: \mathtt{prod}[L](\{i \hookrightarrow \tau_i'\}_{i \in L})} \tag{5.8e}$$

$$\frac{\{\Omega \vdash \tau_i <: \tau_i'\}_{i \in L}}{\Omega \vdash \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) <: \mathtt{sum}[L](\{i \hookrightarrow \tau_i'\}_{i \in L})} \tag{5.8f}$$

Expression typing TODO: describe these

$$\frac{\Omega \vdash e : \tau \qquad \Omega \vdash \tau <: \tau'}{\Omega \vdash e : \tau'} \tag{5.9a}$$

$$\frac{}{\Omega, x : \tau \vdash x : \tau} \tag{5.9b}$$

$$\frac{\Omega \vdash \tau \ \mathsf{type} \qquad \Omega, x : \tau \vdash e : \tau'}{\Omega \vdash \mathtt{lam}\{\tau\}(x.e) : \mathtt{parr}(\tau; \tau')} \tag{5.9c}$$

$$\frac{\Omega \vdash e_1 : \mathtt{parr}(\tau; \tau') \qquad \Omega \vdash e_2 : \tau}{\Omega \vdash \mathtt{ap}(e_1; e_2) : \tau'} \tag{5.9d}$$

$$\frac{\Omega \vdash \kappa \ \mathsf{kind} \qquad \Omega, u :: \kappa \vdash e : \tau}{\Omega \vdash \mathtt{clam}\{\kappa\}(u.e) : \mathtt{all}\{\kappa\}(u.\tau)} \tag{5.9e}$$

$$\frac{\Omega \vdash e : \mathtt{all}\{\kappa\}(u.\tau) \qquad \Omega \vdash c :: \kappa}{\Omega \vdash \mathtt{cap}\{c\}(e) : [c/u]\tau} \tag{5.9f}$$

$$\frac{\Omega, t :: \mathtt{Type} \vdash \tau \ \mathsf{type} \qquad \Omega \vdash e : [\mathtt{rec}(t.\tau)/t]\tau}{\Omega \vdash \mathtt{fold}\{t.\tau\}(e) : \mathtt{rec}(t.\tau)} \tag{5.9g}$$

$$\frac{\Omega \vdash e : \mathtt{rec}(t.\tau)}{\Omega \vdash \mathtt{unfold}(e) : [\mathtt{rec}(t.\tau)/t]\tau} \tag{5.9h}$$

$$\frac{\{\Omega \vdash e_i : \tau_i\}_{i \in L}}{\Omega \vdash \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) : \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{5.9i}$$

$$\frac{\Omega \vdash e : \mathtt{prod}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)}{\Omega \vdash \mathtt{pr}[\ell](e) : \tau} \tag{5.9j}$$

$$\frac{\{\Omega \vdash \tau_i \ \mathsf{type}\}_{i \in L} \qquad \Omega \vdash \tau \ \mathsf{type} \qquad \Omega \vdash e : \tau}{\Omega \vdash \mathtt{in}[L, \ell][\ell]\{\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau\}(e) : \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)} \tag{5.9k}$$

$$\frac{\Omega \vdash e : \tau \qquad \Omega \vdash \tau' \ \mathsf{type} \qquad \{\Omega \vdash r_i : \tau \Rightarrow \tau'\}_{1 \leq i \leq n}}{\Omega \vdash \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \leq i \leq n}) : \tau'} \tag{5.9l}$$

$$\frac{\Omega \vdash M \ \mathsf{mval} \qquad \Omega \vdash M : \mathtt{sig}\{\kappa\}(u.\tau)}{\Omega \vdash \mathtt{val}(M) : [\mathtt{con}(M)/u]\tau} \tag{5.9m}$$

maybe last one needs self-recognition?

Rule typing

$$\frac{\Omega \vdash p : \tau \dashv\mid \Omega' \qquad \Omega \cup \Omega' \vdash e : \tau'}{\Omega \vdash \mathtt{rule}(p.e) : \tau \Rightarrow \tau'} \tag{5.10}$$

Pattern typing

$$\frac{\Omega \vdash p : \tau \dashv\mid \Omega' \qquad \Omega \vdash \tau <: \tau'}{\Omega \vdash p : \tau' \dashv\mid \Omega'} \tag{5.11a}$$

$$\frac{}{\Omega \vdash x : \tau \dashv\mid x : \tau} \tag{5.11b}$$

$$\frac{}{\Omega \vdash \mathtt{wildp} : \tau \dashv\mid \emptyset} \tag{5.11c}$$

$$\frac{\Omega \vdash p : [\mathtt{rec}(t.\tau)/t]\tau \dashv\mid \Omega'}{\Omega \vdash \mathtt{foldp}(p) : \mathtt{rec}(t.\tau) \dashv\mid \Omega'} \tag{5.11d}$$

$$\frac{\{\Omega \vdash p_i : \tau_i \dashv\mid \Omega_i\}_{i \in L}}{\Omega \vdash \mathtt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\mid \cup_{i \in L} \Omega_i} \tag{5.11e}$$

$$\frac{\Omega \vdash p : \tau \dashv\mid \Omega'}{\Omega \vdash \mathtt{inp}[\ell](p) : \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\mid \Omega'} \tag{5.11f}$$

Signatures and Structures

Signature formation

$$\frac{\Omega \vdash \kappa \ \mathsf{kind} \qquad \Omega, u :: \kappa \vdash \tau \ \mathsf{type}}{\Omega \vdash \mathtt{sig}\{\kappa\}(u.\tau) \ \mathsf{sig}} \tag{5.12}$$

Signature equality

$$\frac{\Omega \vdash \kappa \equiv \kappa' \qquad \Omega, u :: \kappa \vdash \tau \equiv \tau' \ \mathsf{type}}{\Omega \vdash \mathtt{sig}\{\kappa\}(u.\tau) \equiv \mathtt{sig}\{\kappa'\}(u.\tau')} \tag{5.13}$$

Subsignature

$$\frac{\Omega \vdash \kappa <:: \kappa' \qquad \Omega, u :: \kappa \vdash \tau <: \tau'}{\Omega \vdash \mathtt{sig}\{\kappa\}(u.\tau) <: \mathtt{sig}\{\kappa'\}(u.\tau')} \tag{5.14}$$

Signature matching

$$\frac{\Omega \vdash M : \sigma \qquad \Omega \vdash \sigma <: \sigma'}{\Omega \vdash M : \sigma'} \tag{5.15a}$$

$$\frac{}{\Omega, X : \sigma \vdash X : \sigma} \tag{5.15b}$$

$$\frac{\Omega \vdash c :: \kappa \qquad \Omega \vdash e : [c/u]\tau}{\Omega \vdash \mathtt{struct}(c; e) : \mathtt{sig}\{\kappa\}(u.\tau)} \tag{5.15c}$$

$$\frac{\Omega \vdash \sigma \ \mathsf{sig} \qquad \Omega \vdash M : \sigma}{\Omega \vdash \mathtt{seal}\{\sigma\}(M) : \sigma} \tag{5.15d}$$

$$\frac{\Omega \vdash M : \sigma \qquad \Omega \vdash \sigma' \ \mathsf{sig} \qquad \Omega, X : \sigma \vdash M' : \sigma'}{\Omega \vdash \mathtt{mlet}\{\sigma'\}(M; X.M') : \sigma'} \tag{5.15e}$$

Module values

$$\frac{}{\Omega \vdash \mathtt{struct}(c; e) \ \mathsf{mval}} \tag{5.16a}$$

$$\frac{}{\Omega, X : \sigma \vdash X \ \mathsf{mval}} \tag{5.16b}$$

TODO: metatheory of statics of inner core

5.2.3 Structural Dynamics

Expressions

Judgement Form	**Description**
$e \mapsto e'$	e transitions to e'
$\Omega \vdash e \ \mathsf{val}$	e is a value

Modules

Judgement Form	**Description**
$M \mapsto M'$	M transitions to M'
$\Omega \vdash M \ \mathsf{mval}$	M is a module value

5.2.4 Syntax of the Surface Language

Syntax of the Surface Core Language

Unexpanded kinds and constructors in Figure 5.4

Unexpanded expressions, rules and patterns in Figure 5.5

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

Figure 5.4: Syntax of unexpanded kinds and constructors in $\mathsf{miniVerse}_\forall$. 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
			$\texttt{upr}[\ell](\hat{e})$	$\hat{e} \cdot \ell$	projection
			$\texttt{uin}[\ell](\hat{e})$	$\texttt{inj}\ \ell \cdot \hat{e}$	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
			$\texttt{uinp}[\ell](\hat{p})$	$\texttt{inj}\ \ell \cdot \hat{p}$	injection pattern
			$\texttt{uappptsm}[b]\{\hat{e}\}$	$\hat{e}\ /b/$	ppTSM application
			$\texttt{uplit}[b]$	$/b/$	ppTSM unadorned literal

Figure 5.5: Abstract syntax of unexpanded expressions, rules and patterns in miniVerse$_\forall$.

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{`$\hat{\epsilon}$`}(`$\hat{a}.\hat{M}$`)`	`let syntax` $\hat{a} = \hat{\epsilon}$ `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{`$\hat{\epsilon}$`}(`$\hat{a}.\hat{M}$`)`	`let syntax` $\hat{a} = \hat{\epsilon}$ `for patterns in` $\hat{M}$	ppTSM binding
			...	...	ppTSM designation

Figure 5.6: Abstract syntax of unexpanded module expressions and signatures in miniVerse$_\forall$.

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 5.7: Abstract syntax of TSM types, unexpanded TSM types, TSM expressions and unexpanded TSM expressions in miniVerse$_\forall$.

Syntax of the TSM Language

Macro types, unexpanded macro types, macro expressions and unexpanded macro expressions in Figure 5.7

Syntax of Type Patterns

TODO: do this

Syntax of the Surface Module Language

Figure 5.6

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
$\hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\vdash \hat{\Omega}$	$\hat{p}$ 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

$$\frac{\hat{\Omega} \vdash \hat{\kappa}_1 \rightsquigarrow \kappa_1 \ \mathsf{kind} \qquad \hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa_1 \vdash \hat{\kappa}_2 \rightsquigarrow \kappa_2 \ \mathsf{kind}}{\hat{\Omega} \vdash \mathtt{udarr}(\hat{\kappa}_1; \hat{u}.\hat{\kappa}_2) \rightsquigarrow \mathtt{darr}(\kappa_1; u.\kappa_2) \ \mathsf{kind}} \tag{5.17a}$$

$$\frac{}{\hat{\Omega} \vdash \mathtt{uunit} \rightsquigarrow \mathtt{unit} \ \mathsf{kind}} \tag{5.17b}$$

$$\frac{\hat{\Omega} \vdash \hat{\kappa}_1 \rightsquigarrow \kappa_1 \ \mathsf{kind} \qquad \hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa_1 \vdash \hat{\kappa}_2 \rightsquigarrow \kappa_2 \ \mathsf{kind}}{\hat{\Omega} \vdash \mathtt{udprod}(\hat{\kappa}_1; \hat{u}.\hat{\kappa}_2) \rightsquigarrow \mathtt{dprod}(\kappa_1; u.\kappa_2) \ \mathsf{kind}} \tag{5.17c}$$

$$\frac{}{\hat{\Omega} \vdash \mathtt{uType} \rightsquigarrow \mathtt{Type} \ \mathsf{kind}} \tag{5.17d}$$

$$\frac{\hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau \Leftarrow \mathtt{Type}}{\hat{\Omega} \vdash \mathtt{uS}(\hat{\tau}) \rightsquigarrow \mathtt{S}(\tau) \ \mathsf{kind}} \tag{5.17e}$$

Synthetic constructor expansion

$$\frac{}{\hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa \vdash \hat{u} \rightsquigarrow u \Rightarrow \kappa} \tag{5.18a}$$

$$\frac{\hat{\Omega} \vdash \hat{\kappa} \rightsquigarrow \kappa \;\mathsf{kind} \qquad \hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Leftarrow \kappa}{\hat{\Omega} \vdash \mathtt{uasc\{}\hat{\kappa}\mathtt{\}(}\hat{c}\mathtt{)} \rightsquigarrow c \Rightarrow \kappa} \tag{5.18b}$$

$$\frac{\hat{\Omega} \vdash \hat{c}_1 \rightsquigarrow c_1 \Rightarrow \mathtt{darr(}\kappa_2\mathtt{;}u.\kappa\mathtt{)} \qquad \hat{\Omega} \vdash \hat{c}_2 \rightsquigarrow c_2 \Leftarrow \kappa_2}{\hat{\Omega} \vdash \mathtt{uapp(}\hat{c}_1\mathtt{;}\hat{c}_2\mathtt{)} \rightsquigarrow \mathtt{app(}c_1\mathtt{;}c_2\mathtt{)} \Rightarrow [c_1/u]\kappa} \tag{5.18c}$$

$$\frac{}{\hat{\Omega} \vdash \mathtt{utriv} \rightsquigarrow \mathtt{triv} \Rightarrow \mathtt{unit}} \tag{5.18d}$$

$$\frac{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Rightarrow \mathtt{dprod(}\kappa_1\mathtt{;}u.\kappa_2\mathtt{)}}{\hat{\Omega} \vdash \mathtt{uprl(}\hat{c}\mathtt{)} \rightsquigarrow \mathtt{prl(}c\mathtt{)} \Rightarrow \kappa_1} \tag{5.18e}$$

$$\frac{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Rightarrow \mathtt{dprod(}\kappa_1\mathtt{;}u.\kappa_2\mathtt{)}}{\hat{\Omega} \vdash \mathtt{uprr(}\hat{c}\mathtt{)} \rightsquigarrow \mathtt{prr(}c\mathtt{)} \Rightarrow [\mathtt{prl(}c\mathtt{)}/u]\kappa_2} \tag{5.18f}$$

$$\frac{\hat{\Omega} \vdash \hat{\tau}_1 \rightsquigarrow \tau_1 \Leftarrow \mathtt{Type} \qquad \hat{\Omega} \vdash \hat{\tau}_2 \rightsquigarrow \tau_2 \Leftarrow \mathtt{Type}}{\hat{\Omega} \vdash \mathtt{uparr(}\hat{\tau}_1\mathtt{;}\hat{\tau}_2\mathtt{)} \rightsquigarrow \mathtt{parr(}\tau_1\mathtt{;}\tau_2\mathtt{)} \Rightarrow \mathtt{Type}} \tag{5.18g}$$

$$\frac{\hat{\Omega} \vdash \hat{\kappa} \rightsquigarrow \kappa \;\mathsf{kind} \qquad \hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa \vdash \hat{\tau} \rightsquigarrow \tau \Leftarrow \mathtt{Type}}{\hat{\Omega} \vdash \mathtt{uall\{}\hat{\kappa}\mathtt{\}(}\hat{u}.\hat{\tau}\mathtt{)} \rightsquigarrow \mathtt{all\{}\kappa\mathtt{\}(}u.\tau\mathtt{)} \Rightarrow \mathtt{Type}} \tag{5.18h}$$

$$\frac{\hat{\Omega}, \hat{t} \rightsquigarrow t :: \mathtt{Type} \vdash \hat{\tau} \rightsquigarrow \tau \Leftarrow \mathtt{Type}}{\hat{\Omega} \vdash \mathtt{urec(}\hat{t}.\hat{\tau}\mathtt{)} \rightsquigarrow \mathtt{rec(}t.\tau\mathtt{)} \Rightarrow \mathtt{Type}} \tag{5.18i}$$

$$\frac{\{\hat{\Omega} \vdash \hat{\tau}_i \rightsquigarrow \tau_i \Leftarrow \mathtt{Type}\}_{1 \leq i \leq n}}{\hat{\Omega} \vdash \mathtt{uprod}[L](\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}) \rightsquigarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \Rightarrow \mathtt{Type}} \tag{5.18j}$$

$$\frac{\{\hat{\Omega} \vdash \hat{\tau}_i \rightsquigarrow \tau_i \Leftarrow \mathtt{Type}\}_{1 \leq i \leq n}}{\hat{\Omega} \vdash \mathtt{usum}[L](\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}) \rightsquigarrow \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \Rightarrow \mathtt{Type}} \tag{5.18k}$$

$$\frac{}{\hat{\Omega}, \hat{X} \rightsquigarrow X : \mathtt{sig\{}\kappa\mathtt{\}(}u.\tau\mathtt{)} \vdash \mathtt{ucon(}\hat{X}\mathtt{)} \rightsquigarrow \mathtt{con(}X\mathtt{)} \Rightarrow \kappa} \tag{5.18l}$$

Analytic constructor expansion

$$\frac{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Rightarrow \kappa_1 \qquad \Omega \vdash \kappa_1 <:: \kappa_2}{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Leftarrow \kappa_2} \tag{5.19a}$$

$$\frac{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Leftarrow \mathtt{Type}}{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Leftarrow \mathtt{S(}c\mathtt{)}} \tag{5.19b}$$

$$\frac{\hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa_1 \vdash \hat{c}_2 \rightsquigarrow c_2 \Leftarrow \kappa_2}{\hat{\Omega} \vdash \mathtt{uabs(}\hat{u}.\hat{c}_2\mathtt{)} \rightsquigarrow \mathtt{abs(}u.c_2\mathtt{)} \Leftarrow \mathtt{darr(}\kappa_1\mathtt{;}u.\kappa_2\mathtt{)}} \tag{5.19c}$$

$$\frac{\hat{\Omega} \vdash \hat{c}_1 \rightsquigarrow c_1 \Leftarrow \kappa_1 \qquad \hat{\Omega} \vdash \hat{c}_2 \rightsquigarrow c_2 \Leftarrow [c_1/u]\kappa_2}{\hat{\Omega} \vdash \mathtt{upair(}\hat{c}_1\mathtt{;}\hat{c}_2\mathtt{)} \rightsquigarrow \mathtt{pair(}c_1\mathtt{;}c_2\mathtt{)} \Leftarrow \mathtt{dprod(}\kappa_1\mathtt{;}u.\kappa_2\mathtt{)}} \tag{5.19d}$$

Types, Expressions, Rules and Patterns

Type expansion

$$\frac{\hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau \Leftarrow \mathsf{Type}}{\hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type}} \tag{5.20}$$

Synthetic typed expression expansion

$$\frac{}{\Omega, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{x} \rightsquigarrow x \Rightarrow \tau} \tag{5.21a}$$

$$\frac{\hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uasc}\{\hat{\tau}\}(\hat{e}) \rightsquigarrow e \Rightarrow \tau} \tag{5.21b}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \hat{\Omega}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \rightsquigarrow e' \Rightarrow \tau'}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uletval}(\hat{e}; \hat{x}.\hat{e}') \rightsquigarrow \mathtt{ap}(\mathtt{lam}\{\tau\}(x.e'); e) \Rightarrow \tau'} \tag{5.21c}$$

$$\frac{\hat{\Omega} \vdash \hat{\tau}_1 \rightsquigarrow \tau_1\ \mathsf{type} \qquad \hat{\Omega}, \hat{x} \rightsquigarrow x : \tau_1 \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau_2}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{ulam}\{\hat{\tau}_1\}(\hat{x}.\hat{e}) \rightsquigarrow \mathtt{lam}\{\tau_1\}(x.e) \Rightarrow \mathtt{parr}(\tau_1; \tau_2)} \tag{5.21d}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_1 \rightsquigarrow e_1 \Rightarrow \mathtt{parr}(\tau_2; \tau) \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_2 \rightsquigarrow e_2 \Leftarrow \tau_2}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uap}(\hat{e}_1; \hat{e}_2) \rightsquigarrow \mathtt{ap}(e_1; e_2) \Rightarrow \tau} \tag{5.21e}$$

$$\frac{\hat{\Omega} \vdash \hat{\kappa} \rightsquigarrow \kappa\ \mathsf{kind} \qquad \hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uclam}\{\hat{\kappa}\}(\hat{u}.\hat{e}) \rightsquigarrow \mathtt{clam}\{\kappa\}(u.e) \Rightarrow \mathtt{all}\{\kappa\}(u.\tau)} \tag{5.21f}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \mathtt{all}\{\kappa\}(u.\tau) \qquad \hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Rightarrow \kappa}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{ucap}\{\hat{c}\}(\hat{e}) \rightsquigarrow \mathtt{cap}\{c\}(e) \Rightarrow [c/t]\tau} \tag{5.21g}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \mathtt{rec}(t.\tau)}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uunfold}(\hat{e}) \rightsquigarrow \mathtt{unfold}(e) \Rightarrow [\mathtt{rec}(t.\tau)/t]\tau} \tag{5.21h}$$

$$\frac{\hat{e} = \mathtt{utpl}[L](\{i \hookrightarrow \hat{e}_i\}_{i \in L}) \qquad e = \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) \qquad \{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_i \rightsquigarrow e_i \Rightarrow \tau_i\}_{i \in L}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{5.21i}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \mathtt{prod}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{upr}[\ell](\hat{e}) \rightsquigarrow \mathtt{pr}[\ell](e) \Rightarrow \tau} \tag{5.21j}$$

$$\frac{n > 0 \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \rightsquigarrow r_i \Rightarrow \tau \Rightarrow \tau'\}_{1 \leq i \leq n}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{umatch}[n](\hat{e}; \{\hat{r}_i\}_{1 \leq i \leq n}) \rightsquigarrow \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \leq i \leq n}) \Rightarrow \tau'} \tag{5.21k}$$

$$\frac{}{\hat{\Omega}, \hat{X} \rightsquigarrow X : \mathtt{sig}\{\kappa\}(u.\tau) \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uval}(\hat{X}) \rightsquigarrow \mathtt{val}(X) \Rightarrow [\mathtt{con}(X)/u]\tau} \tag{5.21l}$$

$$\frac{\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho \ \mathsf{tsmty} \qquad \varnothing \vdash e_{\mathsf{parse}} : \mathtt{parr}(\mathtt{Body}; \mathtt{ParseResult}(\mathtt{PCEExp})) \qquad \hat{\Omega} \vdash_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \mathtt{defref}[a]; \Psi, a \hookrightarrow \mathtt{petsm}(\rho; e_{\mathsf{parse}}); \mathcal{I} \rangle; \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau}{\hat{\Omega} \vdash_{\langle \mathcal{A}; \Psi; \mathcal{I} \rangle; \hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for\ \ expressions}\ \{e_{\mathsf{parse}}\}\ \mathtt{in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau} \tag{5.21m}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\langle \mathcal{A}; \Psi; \mathcal{I} \rangle} \hat{e} \rightsquigarrow \epsilon \mathbin{@} \rho \qquad \hat{\Omega} \vdash_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \epsilon; \Psi; \mathcal{I} \rangle; \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau}{\hat{\Omega} \vdash_{\langle \mathcal{A}; \Psi; \mathcal{I} \rangle; \hat{\Phi}} \mathtt{let\ syntax}\ \hat{a} = \hat{e}\ \mathtt{for\ expressions\ in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau} \tag{5.21n}$$

$$\frac{\begin{array}{c}\hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega_{\mathsf{app}} \rangle \qquad \hat{\Psi} = \langle \mathcal{A}; \Psi, a \hookrightarrow \mathtt{petsm}(\rho; e_{\mathsf{parse}}); \mathcal{I} \rangle \\ \hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{e} \rightsquigarrow \epsilon \mathbin{@} \mathtt{type}(\tau_{\mathsf{final}}) \qquad \Omega_{\mathsf{app}} \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \Downarrow \epsilon_{\mathsf{normal}} \\ \mathsf{tsmdef}(\epsilon_{\mathsf{normal}}) = a \qquad b \downarrow e_{\mathsf{body}} \qquad e_{\mathsf{parse}}(e_{\mathsf{body}}) \Downarrow \mathtt{inj}\ \mathtt{Success} \cdot e_{\mathsf{pcand}} \\ e_{\mathsf{pcand}} \uparrow_{\mathsf{PCEExp}} \check{e} \\ \Omega_{\mathsf{app}} \vdash^{\mathsf{Exp}}_{\Psi, a \hookrightarrow \mathtt{petsm}(\rho; e_{\mathsf{parse}})} \check{e} \leftarrow\!\!\!\!\!\!\rightarrow_{\epsilon_{\mathsf{normal}}} \grave{e}\ ?\ \mathtt{type}(\tau_{\mathsf{cand}}) \dashv \omega : \Omega_{\mathsf{params}} \\ \Omega_{\mathsf{params}} \vdash^{\hat{\Omega}; \hat{\Psi}; \hat{\Phi}; b} \grave{e} \rightsquigarrow e \Leftarrow \tau_{\mathsf{cand}}\end{array}}{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e}\ /b/ \rightsquigarrow [\omega]e \Rightarrow [\omega]\tau_{\mathsf{cand}}} \tag{5.21o}$$

TODO: peTSM implicit designation

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \mathtt{uetsm}(\tau; e_{\mathsf{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle; \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \mathtt{uetsm}(\tau; e_{\mathsf{parse}}); \mathcal{I} \rangle; \hat{\Phi}} \mathtt{implicit\ syntax}\ \hat{a}\ \mathtt{for\ expressions\ in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \tag{5.21p}$$

$$\frac{\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho \ \mathsf{tsmty} \qquad \varnothing \vdash e_{\mathsf{parse}} : \mathtt{parr}(\mathtt{Body}; \mathtt{ParseResult}(\mathtt{CEPat})) \qquad \hat{\Omega} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \hookrightarrow \mathtt{defref}[a]; \Phi, a \hookrightarrow \mathtt{pptsm}(\rho; e_{\mathsf{parse}}); \mathcal{I} \rangle} \hat{e} \rightsquigarrow e \Rightarrow \tau}{\hat{\Omega} \vdash_{\hat{\Psi}; \langle \mathcal{A}; \Phi; \mathcal{I} \rangle} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for\ \ patterns}\ \{e_{\mathsf{parse}}\}\ \mathtt{in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau} \tag{5.21q}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\langle \mathcal{A}; \Phi; \mathcal{I} \rangle} \hat{e} \rightsquigarrow \epsilon \mathbin{@} \rho \qquad \hat{\Omega} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \hookrightarrow \epsilon; \Phi; \mathcal{I} \rangle} \hat{e} \rightsquigarrow e \Rightarrow \tau}{\hat{\Omega} \vdash_{\hat{\Psi}; \langle \mathcal{A}; \Phi; \mathcal{I} \rangle} \mathtt{let\ syntax}\ \hat{a} = \hat{e}\ \mathtt{for\ patterns\ in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau} \tag{5.21r}$$

TODO: ppTSM implicit designation

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \mathtt{uptsm}(\tau; e_{\mathsf{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle} \hat{e} \rightsquigarrow e \Rightarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \mathtt{uetsm}(\tau; e_{\mathsf{parse}}); \mathcal{I} \rangle} \mathtt{implicit\ syntax}\ \hat{a}\ \mathtt{for\ patterns\ in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \tag{5.21s}$$

Analytic typed expression expansion

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \Omega \vdash \tau <: \tau'}{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{5.22a}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \hat{\Omega}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \rightsquigarrow e' \Leftarrow \tau'}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uletval}(\hat{e}; \hat{x}.\hat{e}') \rightsquigarrow \mathtt{ap}(\mathtt{lam}\{\tau\}(x.e'); e) \Leftarrow \tau'} \tag{5.22b}$$

$$\frac{\hat{\Omega}, \hat{x} \rightsquigarrow x : \tau_1 \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau_2}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uanalam}(\hat{x}.\hat{e}) \rightsquigarrow \mathtt{lam}\{\tau_1\}(x.e) \Leftarrow \mathtt{parr}(\tau_1; \tau_2)} \tag{5.22c}$$

$$\frac{\hat{\Delta}, \hat{t} \rightsquigarrow t \ \mathsf{type}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{utlam}(\hat{t}.\hat{e}) \rightsquigarrow \mathtt{tlam}(t.e) \Leftarrow \mathtt{all}(t.\tau)} \tag{5.22d}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow [\mathtt{rec}(t.\tau)/t]\tau}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{ufold}(\hat{e}) \rightsquigarrow \mathtt{fold}\{t.\tau\}(e) \Leftarrow \mathtt{rec}(t.\tau)} \tag{5.22e}$$

$$\frac{\hat{e} = \mathtt{utpl}[L](\{i \hookrightarrow \hat{e}_i\}_{i \in L}) \qquad e = \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) \qquad \{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_i \rightsquigarrow e_i \Leftarrow \tau_i\}_{i \in L}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{5.22f}$$

$$\frac{\hat{e} = \mathtt{uin}[\ell](\hat{e}') \qquad e = \mathtt{in}[L, \ell][\ell]\{\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau\}(e') \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \rightsquigarrow e' \Leftarrow \tau}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)} \tag{5.22g}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \rightsquigarrow r_i \Leftarrow \tau \Mapsto \tau'\}_{1 \leq i \leq n}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{umatch}[n](\hat{e}; \{\hat{r}_i\}_{1 \leq i \leq n}) \rightsquigarrow \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \leq i \leq n}) \Leftarrow \tau'} \tag{5.22h}$$

$$\frac{\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho \ \mathsf{tsmty} \qquad \emptyset \vdash e_{\mathrm{parse}} : \mathtt{parr}(\mathtt{Body}; \mathtt{ParseResult}(\mathtt{PCEExp})) \qquad \hat{\Omega} \vdash_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \mathtt{defref}[a]; \Psi, a \hookrightarrow \mathtt{petsm}(\rho; e_{\mathrm{parse}}); \mathcal{I} \rangle; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Omega} \vdash_{\langle \mathcal{A}; \Psi; \mathcal{I} \rangle; \hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for}\ \ \mathtt{expressions}\ \{e_{\mathrm{parse}}\}\ \mathtt{in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau} \tag{5.22i}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\langle \mathcal{A}; \Psi; \mathcal{I} \rangle} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \rho \qquad \hat{\Omega} \vdash_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \epsilon; \Psi; \mathcal{I} \rangle; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Omega} \vdash_{\langle \mathcal{A}; \Psi; \mathcal{I} \rangle; \hat{\Phi}} \mathtt{let\ syntax}\ \hat{a} = \hat{\epsilon}\ \mathtt{for\ expressions\ in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau} \tag{5.22j}$$

TODO: peTSM implicit designation

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \mathtt{uetsm}(\tau; e_{\mathrm{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \mathtt{uetsm}(\tau; e_{\mathrm{parse}}); \mathcal{I} \rangle; \hat{\Phi}} \mathtt{implicit\ syntax}\ \hat{a}\ \mathtt{for\ expressions\ in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{5.22k}$$

TODO: peTSM implicit application

$$\frac{b \downarrow e_{\mathrm{body}} \qquad e_{\mathrm{parse}}(e_{\mathrm{body}}) \Downarrow \mathtt{inj}\ \mathtt{Success} \cdot e_{\mathrm{cand}} \qquad e_{\mathrm{cand}} \uparrow_{\mathsf{CEExp}} \grave{e} \qquad \emptyset\ \emptyset \vdash^{\hat{\Delta}; \hat{\Gamma}; \langle \mathcal{A}; \Psi, a \hookrightarrow \mathtt{uetsm}(\tau; e_{\mathrm{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle; \hat{\Phi}; b} \grave{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A}; \Psi, a \hookrightarrow \mathtt{uetsm}(\tau; e_{\mathrm{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle; \hat{\Phi}} \mathtt{uelit}[b] \rightsquigarrow e \Leftarrow \tau} \tag{5.22l}$$

$$\frac{\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho \ \mathsf{tsmty} \qquad \varnothing \vdash e_{\mathrm{parse}} : \mathtt{parr}(\mathtt{Body}; \mathtt{ParseResult}(\mathtt{CEPat})) \qquad \hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \mathtt{defref}[a]; \Phi, a \hookrightarrow \mathtt{pptsm}(\rho; e_{\mathrm{parse}}); \mathcal{I} \rangle} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A};\Phi;\mathcal{I} \rangle} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for}\ \ \mathtt{patterns}\ \{e_{\mathrm{parse}}\}\ \mathtt{in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau} \tag{5.22m}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\langle \mathcal{A};\Phi;\mathcal{I} \rangle} \hat{e} \rightsquigarrow \epsilon \mathbin{@} \rho \qquad \hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \epsilon; \Phi; \mathcal{I} \rangle} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A};\Phi;\mathcal{I} \rangle} \mathtt{let\ syntax}\ \hat{a} = \hat{e}\ \mathtt{for\ patterns\ in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau} \tag{5.22n}$$

TODO: ppTSM implicit designation

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \mathtt{uptsm}(\tau; e_{\mathrm{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \mathtt{uetsm}(\tau; e_{\mathrm{parse}}); \mathcal{I} \rangle} \mathtt{implicit\ syntax}\ \hat{a}\ \mathtt{for\ patterns\ in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{5.22o}$$

Synthetic rule expansion

$$\frac{\hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega \rangle \qquad \hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\!\mid \langle \varnothing; \mathcal{G}'; \varnothing; \Omega' \rangle \qquad \langle \mathcal{D}; \mathcal{G} \uplus \mathcal{G}'; \mathcal{M}; \Omega \cup \Omega' \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau'}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{urule}(\hat{p}.\hat{e}) \rightsquigarrow \mathtt{rule}(p.e) \Rightarrow \tau \Mapsto \tau'} \tag{5.23}$$

Analytic rule expansion

$$\frac{\hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega \rangle \qquad \hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\!\mid \langle \varnothing; \mathcal{G}'; \varnothing; \Omega' \rangle \qquad \langle \mathcal{D}; \mathcal{G} \uplus \mathcal{G}'; \mathcal{M}; \Omega \cup \Omega' \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{urule}(\hat{p}.\hat{e}) \rightsquigarrow \mathtt{rule}(p.e) \Leftarrow \tau \Mapsto \tau'} \tag{5.24}$$

Typed pattern expansion

$$\frac{\hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega \rangle \qquad \hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\!\mid \hat{\Omega}' \qquad \Omega \vdash \tau <: \tau'}{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau' \dashv\!\!\mid \hat{\Omega}'} \tag{5.25a}$$

$$\frac{}{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{x} \rightsquigarrow x : \tau \dashv\!\!\mid \langle \varnothing; \hat{x} \rightsquigarrow x; \varnothing; x : \tau \rangle} \tag{5.25b}$$

$$\frac{}{\hat{\Omega} \vdash_{\hat{\Phi}} \mathtt{uwildp} \rightsquigarrow \mathtt{wildp} : \tau \dashv\!\!\mid \langle \varnothing; \varnothing; \varnothing; \varnothing \rangle} \tag{5.25c}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : [\mathtt{rec}(t.\tau)/t]\tau \dashv\!\!\mid \hat{\Omega}'}{\hat{\Omega} \vdash_{\hat{\Phi}} \mathtt{ufoldp}(\hat{p}) \rightsquigarrow \mathtt{foldp}(p) : \mathtt{rec}(t.\tau) \dashv\!\!\mid \hat{\Omega}'} \tag{5.25d}$$

$$\frac{\hat{p} = \mathtt{utplp}[L](\{i \hookrightarrow \hat{p}_i\}_{i \in L}) \qquad p = \mathtt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) \qquad \{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{p}_i \rightsquigarrow p_i : \tau_i \dashv\!\!\mid \hat{\Omega}_i\}_{i \in L}}{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\!\!\mid \cup_{i \in L} \hat{\Omega}_i} \tag{5.25e}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\!\mid \hat{\Omega}'}{\hat{\Omega} \vdash_{\hat{\Phi}} \mathtt{uinp[}\ell\mathtt{](}\hat{p}\mathtt{)} \rightsquigarrow \mathtt{inp[}\ell\mathtt{](}p\mathtt{)} : \mathtt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\!\!\mid \hat{\Omega}'} \tag{5.25f}$$

$$\frac{\begin{array}{c}\hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega_{\text{app}} \rangle \qquad \hat{\Phi} = \langle \mathcal{A}; \Phi, a \hookrightarrow \mathtt{pptsm}(\rho; e_{\text{parse}}); \mathcal{I} \rangle \\ \hat{\Omega} \vdash^{\textsf{Pat}}_{\hat{\Phi}} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \mathtt{type(}\tau_{\text{final}}\mathtt{)} \qquad \mathsf{tsmdef}(\epsilon) = a \\ b \downarrow e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \mathtt{inj}\ \mathtt{Success} \cdot e_{\text{cand}} \qquad e_{\text{cand}} \uparrow_{\textsf{CEPat}} \grave{p} \\ \Omega_{\text{app}} \vdash^{\textsf{Pat}}_{\Phi, a \hookrightarrow \mathtt{pptsm}(\rho; e_{\text{parse}})} \cdot \looparrowright_{\epsilon} \cdot\ ?\ \mathtt{type(}\tau_{\text{cand}}\mathtt{)} \dashv \omega : \Omega_{\text{params}} \\ \Omega_{\text{param}} \vdash^{\hat{\Omega}; \hat{\Phi}; b} \grave{p} \rightsquigarrow p : \tau_{\text{cand}} \dashv\!\!\mid^{\hat{\Omega}'}\end{array}}{\hat{\Omega} \vdash_{\hat{\Phi}} \hat{\epsilon}\ /b/ \rightsquigarrow p : [\omega]\tau_{\text{cand}} \dashv\!\!\mid \hat{\Omega}'} \tag{5.25g}$$

TODO: ppTSM implicit application

$$\frac{\begin{array}{c} b \downarrow e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \mathtt{inj}\ \mathtt{Success} \cdot e_{\text{cand}} \qquad e_{\text{cand}} \uparrow_{\textsf{CEPat}} \grave{p} \\ \Omega_{\text{param}} \vdash^{\Delta; \langle \mathcal{A}; \Phi, a \hookrightarrow \mathtt{uptsm}(\tau; e_{\text{parse}}); \mathcal{I}, \tau \hookrightarrow a \rangle; b} \grave{p} \rightsquigarrow p : \tau \dashv\!\!\mid^{\hat{\Omega}} \end{array}}{\Delta \vdash_{\langle \mathcal{A}; \Phi, a \hookrightarrow \mathtt{uptsm}(\tau; e_{\text{parse}}); \mathcal{I}, \tau \hookrightarrow a \rangle}\ /b/ \rightsquigarrow p : \tau \dashv\!\!\mid \hat{\Omega}} \tag{5.25h}$$

Unexpanded Signatures and Module Expressions

Signature expansion

$$\frac{\hat{\Omega} \vdash \hat{\kappa} \rightsquigarrow \kappa\ \mathsf{kind} \qquad \hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type}}{\hat{\Omega} \vdash \mathtt{usig\{}\hat{\kappa}\mathtt{\}(}\hat{u}.\hat{\tau}\mathtt{)} \rightsquigarrow \mathtt{sig\{}\kappa\mathtt{\}(}u.\tau\mathtt{)}\ \mathsf{sig}} \tag{5.26}$$

Synthetic module expression expansion

$$\frac{}{\hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma \vdash_{\hat{\Phi}; \hat{\Psi}} \hat{X} \rightsquigarrow X \Rightarrow \sigma} \tag{5.27a}$$

$$\frac{\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma\ \mathsf{sig} \qquad \hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{M} \rightsquigarrow M \Leftarrow \sigma}{\hat{\Omega} \vdash_{\hat{\Psi}_{\text{E}}; \hat{\Psi}_{\text{P}}} \mathtt{useal\{}\hat{\sigma}\mathtt{\}(}\hat{M}\mathtt{)} \rightsquigarrow \mathtt{seal\{}\sigma\mathtt{\}(}M\mathtt{)} \Rightarrow \sigma} \tag{5.27b}$$

$$\frac{\begin{array}{c}\hat{\Omega} \vdash_{\hat{\Psi}_{\text{E}}; \hat{\Psi}_{\text{P}}} \hat{M} \rightsquigarrow M \Rightarrow \sigma \qquad \hat{\Omega} \vdash \hat{\sigma}' \rightsquigarrow \sigma'\ \mathsf{sig} \\ \hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{M}' \rightsquigarrow M' \Leftarrow \sigma' \end{array}}{\hat{\Omega} \vdash_{\hat{\Psi}_{\text{E}}; \hat{\Psi}_{\text{P}}} \mathtt{umlet\{}\hat{\sigma}'\mathtt{\}(}\hat{M}; \hat{X}.\hat{M}'\mathtt{)} \rightsquigarrow \mathtt{mlet\{}\sigma'\mathtt{\}(}M; X.M'\mathtt{)} \Rightarrow \sigma'} \tag{5.27c}$$

$$\frac{\begin{array}{c}\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho\ \mathsf{tsmty} \qquad \emptyset \vdash e_{\text{parse}} : \mathtt{parr(Body;ParseResult(PCEExp))} \\ \hat{\Omega} \vdash_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \mathtt{defref[}a\mathtt{]}; \Psi, a \hookrightarrow \mathtt{petsm}(\rho; e_{\text{parse}}); \mathcal{I} \rangle; \hat{\Phi}} \hat{M} \rightsquigarrow M \Rightarrow \sigma \end{array}}{\hat{\Omega} \vdash_{\langle \mathcal{A}; \Psi; \mathcal{I} \rangle; \hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for}\ \mathtt{expressions}\ \{e_{\text{parse}}\}\ \mathtt{in}\ \hat{M} \rightsquigarrow M \Rightarrow \sigma} \tag{5.27d}$$

$$\frac{\hat{\Omega} \vdash^{\textsf{Exp}}_{\langle \mathcal{A}; \Psi; \mathcal{I} \rangle} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \rho \qquad \hat{\Omega} \vdash_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \epsilon; \Psi; \mathcal{I} \rangle; \hat{\Phi}} \hat{M} \rightsquigarrow M \Rightarrow \sigma}{\hat{\Omega} \vdash_{\langle \mathcal{A}; \Psi; \mathcal{I} \rangle; \hat{\Phi}} \mathtt{let}\ \mathtt{syntax}\ \hat{a} = \hat{\epsilon}\ \mathtt{for}\ \mathtt{expressions}\ \mathtt{in}\ \hat{M} \rightsquigarrow M \Rightarrow \sigma} \tag{5.27e}$$

TODO: peTSM implicit designation at module level

$$\frac{\cdots}{\cdots} \tag{5.27f}$$

$$\frac{\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho\ \mathsf{tsmty} \qquad \varnothing \vdash e_{\mathrm{parse}} : \mathtt{parr}(\mathtt{Body};\mathtt{ParseResult}(\mathtt{CEPat})) \quad \hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \mathtt{defref}[a];\Phi, a \hookrightarrow \mathtt{pptsm}(\rho;e_{\mathrm{parse}});\mathcal{I}\rangle} \hat{M} \rightsquigarrow M \Rightarrow \sigma}{\hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A};\Phi;\mathcal{I}\rangle} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for}\ \mathtt{patterns}\ \{e_{\mathrm{parse}}\}\ \mathtt{in}\ \hat{M} \rightsquigarrow M \Rightarrow \sigma} \tag{5.27g}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\langle \mathcal{A};\Phi;\mathcal{I}\rangle} \hat{e} \rightsquigarrow \epsilon\ @\ \rho \qquad \hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \epsilon;\Phi;\mathcal{I}\rangle} \hat{M} \rightsquigarrow M \Rightarrow \sigma}{\hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A};\Phi;\mathcal{I}\rangle} \mathtt{let}\ \mathtt{syntax}\ \hat{a} = \hat{e}\ \mathtt{for}\ \mathtt{patterns}\ \mathtt{in}\ \hat{M} \rightsquigarrow M \Rightarrow \sigma} \tag{5.27h}$$

TODO: ppTSM implicit designation at module level

$$\frac{\cdots}{\cdots} \tag{5.27i}$$

Analytic module expression expansion

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi}_{\mathrm{E}};\hat{\Psi}_{\mathrm{P}}} \hat{M} \rightsquigarrow M \Rightarrow \sigma \qquad \hat{\Omega} \vdash \sigma <: \sigma'}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{M} \rightsquigarrow M \Leftarrow \sigma'} \tag{5.28a}$$

$$\frac{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Leftarrow \kappa \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow [c/u]\tau}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{ustruct}(\hat{c};\hat{e}) \rightsquigarrow \mathtt{struct}(c;e) \Leftarrow \mathtt{sig}\{\kappa\}(u.\tau)} \tag{5.28b}$$

$$\frac{\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho\ \mathsf{tsmty} \qquad \varnothing \vdash e_{\mathrm{parse}} : \mathtt{parr}(\mathtt{Body};\mathtt{ParseResult}(\mathtt{PCEExp})) \quad \hat{\Omega} \vdash_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \mathtt{defref}[a];\Psi, a \hookrightarrow \mathtt{petsm}(\rho;e_{\mathrm{parse}});\mathcal{I}\rangle;\hat{\Phi}} \hat{M} \rightsquigarrow M \Leftarrow \sigma}{\hat{\Omega} \vdash_{\langle \mathcal{A};\Psi;\mathcal{I}\rangle;\hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for}\ \mathtt{expressions}\ \{e_{\mathrm{parse}}\}\ \mathtt{in}\ \hat{M} \rightsquigarrow M \Leftarrow \sigma} \tag{5.28c}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\langle \mathcal{A};\Psi;\mathcal{I}\rangle} \hat{e} \rightsquigarrow \epsilon\ @\ \rho \qquad \hat{\Omega} \vdash_{\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \epsilon;\Psi;\mathcal{I}\rangle;\hat{\Phi}} \hat{M} \rightsquigarrow M \Leftarrow \sigma}{\hat{\Omega} \vdash_{\langle \mathcal{A};\Psi;\mathcal{I}\rangle;\hat{\Phi}} \mathtt{let}\ \mathtt{syntax}\ \hat{a} = \hat{e}\ \mathtt{for}\ \mathtt{expressions}\ \mathtt{in}\ \hat{M} \rightsquigarrow M \Leftarrow \sigma} \tag{5.28d}$$

TODO: peTSM implicit designation at module level

$$\frac{\cdots}{\cdots} \tag{5.28e}$$

$$\frac{\hat{\Omega} \vdash \hat{\rho} \rightsquigarrow \rho\ \mathsf{tsmty} \qquad \varnothing \vdash e_{\mathrm{parse}} : \mathtt{parr}(\mathtt{Body};\mathtt{ParseResult}(\mathtt{CEPat})) \quad \hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \mathtt{defref}[a];\Phi, a \hookrightarrow \mathtt{pptsm}(\rho;e_{\mathrm{parse}});\mathcal{I}\rangle} \hat{M} \rightsquigarrow M \Leftarrow \sigma}{\hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A};\Phi;\mathcal{I}\rangle} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for}\ \mathtt{patterns}\ \{e_{\mathrm{parse}}\}\ \mathtt{in}\ \hat{M} \rightsquigarrow M \Leftarrow \sigma} \tag{5.28f}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\langle \mathcal{A};\Phi;\mathcal{I}\rangle} \hat{e} \rightsquigarrow \epsilon\ @\ \rho \qquad \hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \hookrightarrow \epsilon;\Phi;\mathcal{I}\rangle} \hat{M} \rightsquigarrow M \Leftarrow \sigma}{\hat{\Omega} \vdash_{\hat{\Psi};\langle \mathcal{A};\Phi;\mathcal{I}\rangle} \mathtt{let}\ \mathtt{syntax}\ \hat{a} = \hat{e}\ \mathtt{for}\ \mathtt{patterns}\ \mathtt{in}\ \hat{M} \rightsquigarrow M \Leftarrow \sigma} \tag{5.28g}$$

TODO: ppTSM implicit designation at module level

$$\frac{\cdots}{\cdots} \tag{5.28h}$$

TSM Types and Expressions

TSM Expression Typing

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

peTSM Expression Evaluation

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

+ 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
$\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \rho$	unexpanded peTSM expression $\hat{\epsilon}$ has expansion ϵ and type ρ
$\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \rho$	unexpanded ppTSM expression $\hat{\epsilon}$ has expansion ϵ and type ρ

TSM type formation

$$\frac{\Omega \vdash \tau \text{ type}}{\Omega \vdash \texttt{type(}\tau\texttt{)} \text{ tsmty}} \tag{5.29a}$$

$$\frac{\Omega, t :: \texttt{Type} \vdash \rho \text{ tsmty}}{\Omega \vdash \texttt{alltypes(}t.\rho\texttt{)} \text{ tsmty}} \tag{5.29b}$$

$$\frac{\Omega \vdash \sigma \text{ sig} \qquad \Omega, X : \sigma \vdash \rho \text{ tsmty}}{\Omega \vdash \texttt{allmods\{}\sigma\texttt{\}(}X.\rho\texttt{)} \text{ tsmty}} \tag{5.29c}$$

Unexpanded TSM type expansion

$$\frac{\hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau \text{ type}}{\hat{\Omega} \vdash \texttt{utype(}\hat{\tau}\texttt{)} \rightsquigarrow \texttt{type(}\tau\texttt{)} \text{ tsmty}} \tag{5.30a}$$

$$\frac{\hat{\Omega}, \hat{t} \rightsquigarrow t :: \texttt{Type} \vdash \hat{\rho} \rightsquigarrow \rho \text{ tsmty}}{\hat{\Omega} \vdash \texttt{ualltypes(}\hat{t}.\hat{\rho}\texttt{)} \rightsquigarrow \texttt{alltypes(}t.\rho\texttt{)} \text{ tsmty}} \tag{5.30b}$$

$$\frac{\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma \text{ sig} \qquad \hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma \vdash \hat{\rho} \rightsquigarrow \rho \text{ tsmty}}{\hat{\Omega} \vdash \texttt{uallmods\{}\hat{\sigma}\texttt{\}(}\hat{X}.\hat{\rho}\texttt{)} \rightsquigarrow \texttt{allmods\{}\sigma\texttt{\}(}X.\rho\texttt{)} \text{ tsmty}} \tag{5.30c}$$

peTSM Expression Typing

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi, a \hookrightarrow \texttt{petsm}(\rho; e_{\text{parse}})} \texttt{defref[}a\texttt{]} \mathbin{@} \rho} \tag{5.31a}$$

$$\frac{\Omega, t :: \mathtt{Type} \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} \rho}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{abstype}(t.\epsilon) \mathbin{@} \mathtt{alltypes}(t.\rho)} \tag{5.31b}$$

$$\frac{\Omega \vdash \sigma\ \mathsf{sig} \qquad \Omega, X : \sigma \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} \rho}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{absmod}\{\sigma\}(X.\epsilon) \mathbin{@} \mathtt{allmods}\{\sigma\}(X.\rho)} \tag{5.31c}$$

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} \mathtt{alltypes}(t.\rho) \qquad \Omega \vdash \tau\ \mathsf{type}}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{aptype}\{\tau\}(\epsilon) \mathbin{@} [\tau/t]\rho} \tag{5.31d}$$

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} \mathtt{allmods}\{\sigma\}(X'.\rho) \qquad \Omega \vdash X : \sigma}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{apmod}\{X\}(\epsilon) \mathbin{@} [X/X']\rho} \tag{5.31e}$$

ppTSM Expression Typing

$$\frac{}{\Omega \vdash^{\mathsf{Pat}}_{\Phi, a \hookrightarrow \mathtt{pptsm}(\rho; e_{\mathrm{parse}})} \mathtt{defref}[a] \mathbin{@} \rho} \tag{5.32a}$$

$$\frac{\Omega, t :: \mathtt{Type} \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mathbin{@} \rho}{\Omega \vdash^{\mathsf{Pat}}_{\Phi} \mathtt{abstype}(t.\epsilon) \mathbin{@} \mathtt{alltypes}(t.\rho)} \tag{5.32b}$$

$$\frac{\Omega \vdash \sigma\ \mathsf{sig} \qquad \Omega, X : \sigma \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mathbin{@} \rho}{\Omega \vdash^{\mathsf{Pat}}_{\Phi} \mathtt{absmod}\{\sigma\}(X.\epsilon) \mathbin{@} \mathtt{allmods}\{\sigma\}(X.\rho)} \tag{5.32c}$$

$$\frac{\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mathbin{@} \mathtt{alltypes}(t.\rho) \qquad \Omega \vdash \tau\ \mathsf{type}}{\Omega \vdash^{\mathsf{Pat}}_{\Phi} \mathtt{aptype}\{\tau\}(\epsilon) \mathbin{@} [\tau/t]\rho} \tag{5.32d}$$

$$\frac{\Omega \vdash^{\mathsf{Pat}}_{\Phi} \epsilon \mathbin{@} \mathtt{allmods}\{\sigma\}(X'.\rho) \qquad \Omega \vdash X : \sigma}{\Omega \vdash^{\mathsf{Pat}}_{\Phi} \mathtt{apmod}\{X\}(\epsilon) \mathbin{@} [X/X']\rho} \tag{5.32e}$$

peTSM Expression Expansion

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mathbin{@} \rho}{\langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega \rangle \vdash^{\mathsf{Exp}}_{\langle \mathcal{A}, \hat{a} \hookrightarrow \epsilon; \Psi; \mathcal{I} \rangle} \mathtt{bindref}[\hat{a}] \rightsquigarrow \epsilon \mathbin{@} \rho} \tag{5.33a}$$

$$\frac{\hat{\Omega}, \hat{t} \rightsquigarrow t :: \mathtt{Type} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \rho}{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \mathtt{uabstype}(\hat{t}.\hat{\epsilon}) \rightsquigarrow \mathtt{abstype}(t.\epsilon) \mathbin{@} \mathtt{alltypes}(t.\rho)} \tag{5.33b}$$

$$\frac{\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma\ \mathsf{sig} \qquad \hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \rho}{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \mathtt{uabsmod}\{\hat{\sigma}\}(\hat{X}.\hat{\epsilon}) \rightsquigarrow \mathtt{absmod}\{\sigma\}(X.\epsilon) \mathbin{@} \mathtt{allmods}\{\sigma\}(X.\rho)} \tag{5.33c}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{\epsilon} \rightsquigarrow \epsilon \mathbin{@} \mathtt{alltypes}(t.\rho) \qquad \hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type}}{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \mathtt{uaptype}\{\hat{\tau}\}(\hat{\epsilon}) \rightsquigarrow \mathtt{aptype}\{\tau\}(\epsilon) \mathbin{@} [\tau/t]\rho} \tag{5.33d}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \hat{e} \rightsquigarrow e \mathbin{@} \mathtt{allmods}\{\sigma\}(X'.\rho) \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{X} \rightsquigarrow X \Leftarrow \sigma}{\hat{\Omega} \vdash^{\mathsf{Exp}}_{\hat{\Psi}} \mathtt{uapmod}\{\hat{X}\}(\hat{e}) \rightsquigarrow \mathtt{apmod}\{X\}(e) \mathbin{@} [X/X']\rho} \tag{5.33e}$$

ppTSM expression expansion

$$\frac{\Omega \vdash^{\mathsf{Pat}}_{\Phi} e \mathbin{@} \rho}{\langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega \rangle \vdash^{\mathsf{Pat}}_{\langle \mathcal{A}, \hat{a} \hookrightarrow e; \Phi; \mathcal{I} \rangle} \mathtt{bindref}[\hat{a}] \rightsquigarrow e \mathbin{@} \rho} \tag{5.34a}$$

$$\frac{\hat{\Omega}, \hat{t} \rightsquigarrow t :: \mathsf{Type} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \hat{e} \rightsquigarrow e \mathbin{@} \rho}{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \mathtt{uabstype}(\hat{t}.\hat{e}) \rightsquigarrow \mathtt{abstype}(t.e) \mathbin{@} \mathtt{alltypes}(t.\rho)} \tag{5.34b}$$

$$\frac{\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma \ \mathsf{sig} \qquad \hat{\Omega}, \hat{X} \rightsquigarrow X : \sigma \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \hat{e} \rightsquigarrow e \mathbin{@} \rho}{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \mathtt{uabsmod}\{\hat{\sigma}\}(\hat{X}.\hat{e}) \rightsquigarrow \mathtt{absmod}\{\sigma\}(X.e) \mathbin{@} \mathtt{allmods}\{\sigma\}(X.\rho)} \tag{5.34c}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \hat{e} \rightsquigarrow e \mathbin{@} \mathtt{alltypes}(t.\rho) \qquad \hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau \ \mathsf{type}}{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \mathtt{uaptype}\{\hat{\tau}\}(\hat{e}) \rightsquigarrow \mathtt{aptype}\{\tau\}(e) \mathbin{@} [\tau/t]\rho} \tag{5.34d}$$

$$\frac{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \hat{e} \rightsquigarrow e \mathbin{@} \mathtt{allmods}\{\sigma\}(X'.\rho) \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{X} \rightsquigarrow X \Leftarrow \sigma}{\hat{\Omega} \vdash^{\mathsf{Pat}}_{\hat{\Phi}} \mathtt{uapmod}\{\hat{X}\}(\hat{e}) \rightsquigarrow \mathtt{apmod}\{X\}(e) \mathbin{@} [X/X']\rho} \tag{5.34e}$$

peTSM expression normal forms

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi, a \hookrightarrow \mathtt{petsm}(\rho; e_{\mathrm{parse}})} \mathtt{defref}[a] \ \mathsf{normal}} \tag{5.35a}$$

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{abstype}(t.e) \ \mathsf{normal}} \tag{5.35b}$$

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{absmod}\{\sigma\}(X.e) \ \mathsf{normal}} \tag{5.35c}$$

$$\frac{e \neq \mathtt{abstype}(t.e') \qquad \Omega \vdash^{\mathsf{Exp}}_{\Psi} e \ \mathsf{normal}}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{aptype}\{\tau\}(e) \ \mathsf{normal}} \tag{5.35d}$$

$$\frac{e \neq \mathtt{absmod}\{\sigma\}(X'.e') \qquad \Omega \vdash^{\mathsf{Exp}}_{\Psi} e \ \mathsf{normal}}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{apmod}\{X\}(e) \ \mathsf{normal}} \tag{5.35e}$$

peTSM transitions

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} e \mapsto e'}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{aptype}\{\tau\}(e) \mapsto \mathtt{aptype}\{\tau\}(e')} \tag{5.36a}$$

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{aptype}\{\tau\}(\mathtt{abstype}(t.e)) \mapsto [\tau/t]e} \tag{5.36b}$$

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} e \mapsto e'}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \mathtt{apmod}\{X\}(e) \mapsto \mathtt{apmod}\{X\}(e')} \tag{5.36c}$$

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \texttt{apmod}\{X\}(\texttt{absmod}\{\sigma\}(X'.\epsilon)) \mapsto [X/X']\epsilon} \tag{5.36d}$$

peTSM reflexive, transitive transitions

$$\frac{}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto^* \epsilon} \tag{5.37a}$$

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto \epsilon'}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto^* \epsilon'} \tag{5.37b}$$

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto^* \epsilon' \qquad \Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon' \mapsto^* \epsilon''}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto^* \epsilon''} \tag{5.37c}$$

peTSM normalization

$$\frac{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \mapsto^* \epsilon' \qquad \Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon' \ \mathsf{normal}}{\Omega \vdash^{\mathsf{Exp}}_{\Psi} \epsilon \Downarrow \epsilon'} \tag{5.38}$$

TSM expression definition extraction

$$\mathsf{tsmdef}(\texttt{defref}[a]) = a \tag{5.39a}$$
$$\mathsf{tsmdef}(\texttt{abstype}(t.\epsilon)) = \mathsf{tsmdef}(\epsilon) \tag{5.39b}$$
$$\mathsf{tsmdef}(\texttt{absmod}\{\sigma\}(X.\epsilon)) = \mathsf{tsmdef}(\epsilon) \tag{5.39c}$$
$$\mathsf{tsmdef}(\texttt{aptype}\{\tau\}(\epsilon)) = \mathsf{tsmdef}(\epsilon) \tag{5.39d}$$
$$\mathsf{tsmdef}(\texttt{apmod}\{X\}(\epsilon)) = \mathsf{tsmdef}(\epsilon) \tag{5.39e}$$

5.2.6 Syntax of Candidate Expansions

Figure 5.10 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 $\mathcal{C}(\tau)$, defined in Sec. 3.2.8.

Sort			Operational Form	Stylized Form	Description
PCEExp	$\check{e}$	::=	`ceexp(`$\grave{e}$`)`	$\grave{e}$	ce-expression
			`cebindtype(`$t.\check{e}$`)`	$\Lambda t.\check{e}$	type binding
			`cebindmod(`$X.\check{e}$`)`	$\Lambda X.\check{e}$	module binding

Figure 5.8: Abstract syntax of parameterized candidate expansion expressions in miniVerse$_\forall$. Parameterized candidate expansion expressions are identified up to α-equivalence.

Sort			Operational Form	Stylized Form	Description
CEKind	$\grave{\kappa}$	::=	`cedarr(`$\grave{\kappa}; u.\grave{\kappa}$`)`	$(u :: \grave{\kappa}) \to \grave{\kappa}$	dependent function
			`ceunit`	$\langle\!\langle\rangle\!\rangle$	nullary product
			`cedprod(`$\grave{\kappa}; u.\grave{\kappa}$`)`	$(u :: \grave{\kappa}) \times \grave{\kappa}$	dependent product
			`ceType`	`T`	types
			`ceS(`$\grave{\tau}$`)`	$[{=}\grave{\tau}]$	singleton
			`cesplicedk`$[m; n]$	`splicedk`$\langle m, n\rangle$	spliced
CECon	$\grave{c}, \grave{\tau}$	::=	u	u	constructor variable
			t	t	type variable
			`ceasc{`$\grave{\kappa}$`}(`$\grave{c}$`)`	$\grave{c} :: \grave{\kappa}$	ascription
			`ceabs(`$u.\grave{c}$`)`	$\lambda u.\grave{c}$	abstraction
			`ceapp(`$\grave{c}; \grave{c}$`)`	$\grave{c}(\grave{c})$	application
			`cetriv`	$\langle\!\langle\rangle\!\rangle$	trivial
			`cepair(`$\grave{c}; \grave{c}$`)`	$\langle\!\langle \grave{c}, \grave{c}\rangle\!\rangle$	pair
			`ceprl(`$\grave{c}$`)`	$\grave{c} \cdot \mathtt{l}$	left projection
			`ceprr(`$\grave{c}$`)`	$\grave{c} \cdot \mathtt{r}$	right projection
			`ceparr(`$\grave{\tau}; \grave{\tau}$`)`	$\grave{\tau} \rightharpoonup \grave{\tau}$	partial function
			`ceall{`$\grave{\kappa}$`}(`$u.\grave{\tau}$`)`	$\forall(u :: \grave{\kappa}).\grave{\tau}$	polymorphic
			`cerec(`$t.\grave{\tau}$`)`	$\mu t.\grave{\tau}$	recursive
			`ceprod`$[L]$`(`$\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}$`)`	$\langle \{i \hookrightarrow \grave{\tau}_i\}_{i \in L} \rangle$	labeled product
			`cesum`$[L]$`(`$\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}$`)`	$[\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}]$	labeled sum
			`cecon(`X`)`	$X \cdot \mathtt{c}$	constructor part
			`cesplicedc`$[m; n]$	`splicedc`$\langle m, n\rangle$	spliced

Figure 5.9: Syntax of candidate expansion kinds and constructors in miniVerse$_\forall$. Candidate expansion kinds and constructors are identified up to α-equivalence.

Sort			Operational Form	Stylized Form	Description
CEExp	$\grave{e}$	::=	x	x	variable
			$\texttt{ceasc}\{\grave{\tau}\}(\grave{e})$	$\grave{e} : \grave{\tau}$	ascription
			$\texttt{celetval}(\grave{e}; x.\grave{e})$	$\texttt{let val}\ x = \grave{e}\ \texttt{in}\ \grave{e}$	value binding
			$\texttt{ceanalam}(x.\grave{e})$	$\lambda x.\grave{e}$	abstraction (unannotated)
			$\texttt{celam}\{\grave{\tau}\}(x.\grave{e})$	$\lambda x{:}\grave{\tau}.\grave{e}$	abstraction (annotated)
			$\texttt{ceap}(\grave{e}; \grave{e})$	$\grave{e}(\grave{e})$	application
			$\texttt{ceclam}\{\grave{\kappa}\}(u.\grave{e})$	$\Lambda u{::}\grave{\kappa}.\grave{e}$	constructor abstraction
			$\texttt{cecap}\{\grave{c}\}(\grave{e})$	$\grave{e}[\grave{c}]$	constructor application
			$\texttt{cefold}(\grave{e})$	$\texttt{fold}(\grave{e})$	fold
			$\texttt{ceunfold}(\grave{e})$	$\texttt{unfold}(\grave{e})$	unfold
			$\texttt{cetpl}[L](\{i \hookrightarrow \grave{e}_i\}_{i \in L})$	$\langle \{i \hookrightarrow \grave{e}_i\}_{i \in L} \rangle$	labeled tuple
			$\texttt{cepr}[\ell](\grave{e})$	$\grave{e} \cdot \ell$	projection
			$\texttt{cein}[\ell](\grave{e})$	$\texttt{inj}\ \ell \cdot \grave{e}$	injection
			$\texttt{cematch}[n](\grave{e}; \{\grave{r}_i\}_{1 \leq i \leq n})$	$\texttt{match}\ \grave{e}\ \{\grave{r}_i\}_{1 \leq i \leq n}$	match
			$\texttt{ceval}(X)$	$X \cdot \texttt{v}$	value part
			$\texttt{cesplicede}[m; n]$	$\texttt{splicede}\langle m, n \rangle$	spliced
CERule	$\grave{r}$	::=	$\texttt{cerule}(p.\grave{e})$	$p \Rightarrow \grave{e}$	rule
CEPat	$\grave{p}$	::=	$\texttt{cewildp}$	$\_$	wildcard pattern
			$\texttt{cefoldp}(p)$	$\texttt{fold}(p)$	fold pattern
			$\texttt{cetplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i \in L})$	$\langle \{i \hookrightarrow \grave{p}_i\}_{i \in L} \rangle$	labeled tuple pattern
			$\texttt{ceinp}[\ell](\grave{p})$	$\texttt{inj}\ \ell \cdot \grave{p}$	injection pattern
			$\texttt{cesplicedp}[m; n]$	$\texttt{splicedp}\langle m, n \rangle$	spliced

Figure 5.10: Abstract syntax of candidate expansion types, expressions, rules and patterns in miniVerse$_\forall$. Candidate expansion terms are identified up to α-equivalence.

- Each expanded expression form maps onto a ce-expression form according to the metafunction $\mathcal{C}(e)$, defined as follows:

$$\begin{aligned}
\mathcal{C}(x) &= x \\
\mathcal{C}(\texttt{lam\{}\tau\texttt{\}(}x.e\texttt{)}) &= \texttt{celam\{}\mathcal{C}(\tau)\texttt{\}(}x.\mathcal{C}(e)\texttt{)} \\
\mathcal{C}(\texttt{ap(}e_1;e_2\texttt{)}) &= \texttt{ceap(}\mathcal{C}(e_1);\mathcal{C}(e_2)\texttt{)} \\
\mathcal{C}(\texttt{tlam(}t.e\texttt{)}) &= \texttt{cetlam(}t.\mathcal{C}(e)\texttt{)} \\
\mathcal{C}(\texttt{tap\{}\tau\texttt{\}(}e\texttt{)}) &= \texttt{cetap\{}\mathcal{C}(\tau)\texttt{\}(}\mathcal{C}(e)\texttt{)} \\
\mathcal{C}(\texttt{fold\{}t.\tau\texttt{\}(}e\texttt{)}) &= \texttt{ceasc\{cerec(}t.\mathcal{C}(\tau)\texttt{)\}(cefold(}\mathcal{C}(e)\texttt{))} \\
\mathcal{C}(\texttt{unfold(}e\texttt{)}) &= \texttt{ceunfold(}\mathcal{C}(e)\texttt{)} \\
\mathcal{C}(\texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L})) &= \texttt{cetpl}[L](\{i \hookrightarrow \mathcal{C}(e_i)\}_{i\in L}) \\
\mathcal{C}(\texttt{in}[L][\ell]\{\{i \hookrightarrow \tau_i\}_{i\in L}\}(e)) &= \texttt{ceasc\{cesum}[L](\{i \hookrightarrow \mathcal{C}(\tau_i)\}_{i\in L})\texttt{\}(cein}[\ell](\mathcal{C}(e))) \\
\mathcal{C}(\texttt{match}[n]\{\tau\}(e;\{r_i\}_{1\le i\le n})) &= \texttt{ceasc\{}\mathcal{C}(\tau)\texttt{\}(cematch}[n](\mathcal{C}(e);\{\mathcal{C}(r_i)\}_{1\le i\le n})\texttt{)}
\end{aligned}$$

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

$$\mathcal{C}(\texttt{rule(}p.e\texttt{)}) = \texttt{cerule(}p.\mathcal{C}(e)\texttt{)}$$

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

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

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

$$\frac{}{\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi, a \hookrightarrow \texttt{petsm}(\rho;e_{\text{parse}})} \texttt{ceexp(}\grave{e}\texttt{)} \looparrowright_{\texttt{defref[}a\texttt{]}} \grave{e} \mathbin{?} \rho \dashv \emptyset : \emptyset} \tag{5.40a}$$

$$\frac{\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \check{e} \Rightarrow_{\epsilon} \grave{e} \,?\, \texttt{alltypes}(t.\rho) \dashv \omega : \Omega \qquad t \notin \text{dom}(\Omega_{\text{app}})}{\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \texttt{cebindtype}(t.\check{e}) \Rightarrow_{\texttt{aptype}\{\tau\}(\epsilon)} \grave{e} \,?\, \rho \dashv \omega, \tau/t : \Omega, t :: \texttt{Type}} \tag{5.40b}$$

$$\frac{\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \check{e} \Rightarrow_{\epsilon} \grave{e} \,?\, \texttt{allmods}\{\sigma\}(X.\rho) \dashv \omega : \Omega \qquad X \notin \text{dom}(\Omega_{\text{app}})}{\Omega_{\text{app}} \vdash^{\mathsf{Exp}}_{\Psi} \texttt{cebindmod}(X.\check{e}) \Rightarrow_{\texttt{apmod}\{X'\}(\epsilon)} \grave{e} \,?\, \rho \dashv \omega, X'/X : \Omega, X : \sigma} \tag{5.40c}$$

Candidate Expansion Pattern Deparameterization

$$\frac{}{\Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi, a \hookrightarrow \texttt{pptsm}(\rho; e_{\text{parse}})} \cdot \Rightarrow_{\texttt{defref}[a]} \cdot \,?\, \rho \dashv \emptyset : \emptyset} \tag{5.41a}$$

$$\frac{\Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi} \cdot \Rightarrow_{\epsilon} \cdot \,?\, \texttt{alltypes}(t.\rho) \dashv \omega : \Omega \qquad t \notin \text{dom}(\Omega_{\text{app}})}{\Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi} \cdot \Rightarrow_{\texttt{aptype}\{\tau\}(\epsilon)} \cdot \,?\, \rho \dashv \omega, \tau/t : \Omega, t :: \texttt{Type}} \tag{5.41b}$$

$$\frac{\Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi} \cdot \Rightarrow_{\epsilon} \cdot \,?\, \texttt{allmods}\{\sigma\}(X.\rho) \dashv \omega : \Omega \qquad X \notin \text{dom}(\Omega_{\text{app}})}{\Omega_{\text{app}} \vdash^{\mathsf{Pat}}_{\Phi} \cdot \Rightarrow_{\texttt{apmod}\{X'\}(\epsilon)} \cdot \,?\, \rho \dashv \omega, X'/X : \Omega, X : \sigma} \tag{5.41c}$$

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

Judgement Form	**Description**
$\Omega \vdash^{\mathbb{C}} \grave{\kappa} \rightsquigarrow \kappa$ 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
$\Omega_{\text{param}} \vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : \tau \dashv\mid^{\hat{\Omega}}$	$\grave{p}$ expands to p and matches values of type τ generating assumptions $\hat{\Upsilon}$

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 (4.4m) and (4.6f)), 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

$$\frac{\Omega \vdash^{\mathbb{C}} \dot{\kappa}_1 \rightsquigarrow \kappa_1 \;\mathsf{kind} \qquad \Omega, u :: \kappa_1 \vdash^{\mathbb{C}} \dot{\kappa}_2 \rightsquigarrow \kappa_2 \;\mathsf{kind}}{\Omega \vdash^{\mathbb{C}} \mathtt{cedarr}(\dot{\kappa}_1; u.\dot{\kappa}_2) \rightsquigarrow \mathtt{darr}(\kappa_1; u.\kappa_2)\;\mathsf{kind}} \tag{5.42a}$$

$$\frac{}{\Omega \vdash^{\mathbb{C}} \mathtt{ceunit} \rightsquigarrow \mathtt{unit}\;\mathsf{kind}} \tag{5.42b}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \dot{\kappa}_1 \rightsquigarrow \kappa_1 \;\mathsf{kind} \qquad \Omega, u :: \kappa_1 \vdash^{\mathbb{C}} \dot{\kappa}_2 \rightsquigarrow \kappa_2 \;\mathsf{kind}}{\Omega \vdash^{\mathbb{C}} \mathtt{cedprod}(\dot{\kappa}_1; u.\dot{\kappa}_2) \rightsquigarrow \mathtt{dprod}(\kappa_1; u.\kappa_2)\;\mathsf{kind}} \tag{5.42c}$$

$$\frac{}{\Omega \vdash^{\mathbb{C}} \mathtt{ceType} \rightsquigarrow \mathtt{Type}\;\mathsf{kind}} \tag{5.42d}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \dot{\tau} \rightsquigarrow \tau \Leftarrow \mathtt{Type}}{\Omega \vdash^{\mathbb{C}} \mathtt{ceS}(\dot{\tau}) \rightsquigarrow \mathtt{S}(\tau)\;\mathsf{kind}} \tag{5.42e}$$

$$\frac{\begin{array}{c}\mathsf{parseUKind}(\mathsf{subseq}(b; m; n)) = \hat{\kappa} \qquad \hat{\Omega} \vdash \hat{\kappa} \rightsquigarrow \kappa\;\mathsf{kind} \\ \hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega_{\mathrm{app}} \rangle \qquad \mathrm{dom}(\Omega) \cap \mathrm{dom}(\Omega_{\mathrm{app}}) = \emptyset\end{array}}{\Omega \vdash^{\hat{\Omega}; b} \mathtt{cesplicedk}[m; n] \rightsquigarrow \kappa\;\mathsf{kind}} \tag{5.42f}$$

Synthetic ce-constructor validation

$$\frac{}{\Omega, u :: \kappa \vdash^{\mathbb{C}} u \rightsquigarrow u \Rightarrow \kappa} \tag{5.43a}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \dot{\kappa} \rightsquigarrow \kappa\;\mathsf{kind} \qquad \Omega \vdash^{\mathbb{C}} \dot{c} \rightsquigarrow c \Leftarrow \kappa}{\Omega \vdash^{\mathbb{C}} \mathtt{ceasc}\{\dot{\kappa}\}(\dot{c}) \rightsquigarrow c \Rightarrow \kappa} \tag{5.43b}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \dot{c}_1 \rightsquigarrow c_1 \Rightarrow \mathtt{darr}(\kappa_2; u.\kappa) \qquad \Omega \vdash^{\mathbb{C}} \dot{c}_2 \rightsquigarrow c_2 \Rightarrow \kappa_2}{\Omega \vdash^{\mathbb{C}} \mathtt{ceapp}(\dot{c}_1; \dot{c}_2) \rightsquigarrow \mathtt{app}(c_1; c_2) \Rightarrow [c_1/u]\kappa} \tag{5.43c}$$

$$\frac{}{\Omega \vdash^{\mathbb{C}} \mathtt{cetriv} \rightsquigarrow \mathtt{triv} \Rightarrow \mathtt{unit}} \tag{5.43d}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \dot{c} \rightsquigarrow c \Rightarrow \mathtt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash^{\mathbb{C}} \mathtt{ceprl}(\dot{c}) \rightsquigarrow \mathtt{prl}(c) \Rightarrow \kappa_1} \tag{5.43e}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \dot{c} \rightsquigarrow c \Rightarrow \mathtt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash^{\mathbb{C}} \mathtt{ceprr}(\dot{c}) \rightsquigarrow \mathtt{prr}(c) \Rightarrow [\mathtt{prl}(c)/u]\kappa_2} \tag{5.43f}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \dot{\tau}_1 \rightsquigarrow \tau_1 \Leftarrow \mathtt{Type} \qquad \Omega \vdash^{\mathbb{C}} \dot{\tau}_2 \rightsquigarrow \tau_2 \Leftarrow \mathtt{Type}}{\Omega \vdash^{\mathbb{C}} \mathtt{ceparr}(\dot{\tau}_1; \dot{\tau}_2) \rightsquigarrow \mathtt{parr}(\tau_1; \tau_2) \Rightarrow \mathtt{Type}} \tag{5.43g}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \dot{\kappa} \rightsquigarrow \kappa\;\mathsf{kind} \qquad \Omega, u :: \kappa \vdash^{\mathbb{C}} \dot{\tau} \rightsquigarrow \tau \Leftarrow \mathtt{Type}}{\Omega \vdash^{\mathbb{C}} \mathtt{ceall}\{\dot{\kappa}\}(u.\dot{\tau}) \rightsquigarrow \mathtt{all}\{\kappa\}(u.\tau) \Rightarrow \mathtt{Type}} \tag{5.43h}$$

$$\frac{\Omega, t :: \texttt{Type} \vdash^{\mathbb{C}} \grave{\tau} \rightsquigarrow \tau \Leftarrow \texttt{Type}}{\Omega \vdash^{\mathbb{C}} \texttt{cerec}(t.\grave{\tau}) \rightsquigarrow \texttt{rec}(t.\tau) \Rightarrow \texttt{Type}} \tag{5.43i}$$

$$\frac{\{\Omega \vdash^{\mathbb{C}} \grave{\tau}_i \rightsquigarrow \tau_i \Leftarrow \texttt{Type}\}_{1 \leq i \leq n}}{\Omega \vdash^{\mathbb{C}} \texttt{uprod}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}) \rightsquigarrow \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \Rightarrow \texttt{Type}} \tag{5.43j}$$

$$\frac{\{\Omega \vdash^{\mathbb{C}} \grave{\tau}_i \rightsquigarrow \tau_i \Leftarrow \texttt{Type}\}_{1 \leq i \leq n}}{\Omega \vdash^{\mathbb{C}} \texttt{cesum}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}) \rightsquigarrow \texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \Rightarrow \texttt{Type}} \tag{5.43k}$$

$$\frac{}{\Omega, X : \texttt{sig}\{\kappa\}(u.\tau) \vdash^{\mathbb{C}} \texttt{cecon}(X) \rightsquigarrow \texttt{con}(X) \Rightarrow \kappa} \tag{5.43l}$$

$$\frac{\mathsf{parseUCon}(\mathsf{subseq}(b; m; n)) = \hat{c} \qquad \hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Rightarrow \kappa \qquad \hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega_{\text{app}} \rangle \qquad \mathrm{dom}(\Omega) \cap \mathrm{dom}(\Omega_{\text{app}}) = \emptyset}{\Omega \vdash^{\hat{\Omega}; b} \texttt{cesplicedc}[m; n] \rightsquigarrow c \Rightarrow \kappa} \tag{5.43m}$$

Analytic constructor expansion

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{c} \rightsquigarrow c \Rightarrow \kappa_1 \qquad \Omega \vdash \kappa_1 <:: \kappa_2}{\Omega \vdash^{\mathbb{C}} \grave{c} \rightsquigarrow c \Leftarrow \kappa_2} \tag{5.44a}$$

$$\frac{\hat{\Omega} \vdash \grave{c} \rightsquigarrow c \Leftarrow \texttt{Type}}{\hat{\Omega} \vdash \grave{c} \rightsquigarrow c \Leftarrow \texttt{S}(c)} \tag{5.44b}$$

$$\frac{\Omega, u :: \kappa_1 \vdash^{\mathbb{C}} \grave{c}_2 \rightsquigarrow c_2 \Leftarrow \kappa_2}{\Omega \vdash^{\mathbb{C}} \texttt{ceabs}(u.\grave{c}_2) \rightsquigarrow \texttt{abs}(u.c_2) \Leftarrow \texttt{darr}(\kappa_1; u.\kappa_2)} \tag{5.44c}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{c}_1 \rightsquigarrow c_1 \Leftarrow \kappa_1 \qquad \Omega \vdash^{\mathbb{C}} \grave{c}_2 \rightsquigarrow c_2 \Leftarrow [c_1/u]\kappa_2}{\Omega \vdash^{\mathbb{C}} \texttt{cepair}(\grave{c}_1; \grave{c}_2) \rightsquigarrow \texttt{pair}(c_1; c_2) \Leftarrow \texttt{dprod}(\kappa_1; u.\kappa_2)} \tag{5.44d}$$

$$\frac{\mathsf{parseUCon}(\mathsf{subseq}(b; m; n)) = \hat{c} \qquad \hat{\Omega} \vdash \hat{c} \rightsquigarrow c \Leftarrow \kappa \qquad \hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega_{\text{app}} \rangle \qquad \mathrm{dom}(\Omega) \cap \mathrm{dom}(\Omega_{\text{app}}) = \emptyset}{\Omega \vdash^{\hat{\Omega}; b} \texttt{cesplicedc}[m; n] \rightsquigarrow c \Leftarrow \kappa} \tag{5.44e}$$

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 (7.2) by Rules (7.6) and Rules (7.7), respectively, as follows.

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{\tau} \rightsquigarrow \tau \Leftarrow \texttt{Type}}{\Omega \vdash^{\mathbb{C}} \grave{\tau} \rightsquigarrow \tau\ \mathsf{type}} \tag{5.45}$$

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

$$\frac{}{\Omega, x : \tau \vdash^{\mathbb{E}} x \rightsquigarrow x \Rightarrow \tau} \tag{5.46a}$$

$$\frac{\Omega \vdash^{\mathsf{cs}(\mathbb{E})} \grave{\tau} \rightsquigarrow \tau \ \mathsf{type} \qquad \Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau}{\Omega \vdash^{\mathbb{E}} \mathtt{ceasc}\{\grave{\tau}\}(\grave{e}) \rightsquigarrow e \Rightarrow \tau} \tag{5.46b}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \Omega, x : \tau \vdash^{\mathbb{E}} \grave{e}' \rightsquigarrow e' \Rightarrow \tau'}{\Omega \vdash^{\mathbb{E}} \mathtt{celetval}(\grave{e}; x.\grave{e}') \rightsquigarrow \mathtt{ap}(\mathtt{lam}\{\tau\}(x.e'); e) \Rightarrow \tau'} \tag{5.46c}$$

$$\frac{\Omega \vdash^{\mathsf{cs}(\mathbb{E})} \grave{\tau}_1 \rightsquigarrow \tau_1 \ \mathsf{type} \qquad \Omega, x : \tau_1 \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau_2}{\Omega \vdash^{\mathbb{E}} \mathtt{celam}\{\grave{\tau}_1\}(x.\grave{e}) \rightsquigarrow \mathtt{lam}\{\tau_1\}(x.e) \Rightarrow \mathtt{parr}(\tau_1; \tau_2)} \tag{5.46d}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e}_1 \rightsquigarrow e_1 \Rightarrow \mathtt{parr}(\tau_2; \tau) \qquad \Omega \vdash^{\mathbb{E}} \grave{e}_2 \rightsquigarrow e_2 \Leftarrow \tau_2}{\Omega \vdash^{\mathbb{E}} \mathtt{ceap}(\grave{e}_1; \grave{e}_2) \rightsquigarrow \mathtt{ap}(e_1; e_2) \Rightarrow \tau} \tag{5.46e}$$

$$\frac{\Omega \vdash^{\mathsf{cs}(\mathbb{E})} \grave{\kappa} \rightsquigarrow \kappa \ \mathsf{kind} \qquad \Omega, u :: \kappa \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau}{\Delta\, \Gamma \vdash^{\mathbb{E}} \mathtt{ceclam}\{\grave{\kappa}\}(u.\grave{e}) \rightsquigarrow \mathtt{clam}\{\kappa\}(u.e) \Rightarrow \mathtt{all}\{\kappa\}(u.\tau)} \tag{5.46f}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \mathtt{all}\{\kappa\}(u.\tau) \qquad \Omega \vdash^{\mathsf{cs}(\mathbb{E})} \grave{c} \rightsquigarrow c \Leftarrow \kappa}{\Omega \vdash^{\mathbb{E}} \mathtt{cecap}\{\grave{c}\}(\grave{e}) \rightsquigarrow \mathtt{cap}\{c\}(e) \Rightarrow [c/u]\tau} \tag{5.46g}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \mathtt{rec}(t.\tau)}{\Omega \vdash^{\mathbb{E}} \mathtt{ceunfold}(\grave{e}) \rightsquigarrow \mathtt{unfold}(e) \Rightarrow [\mathtt{rec}(t.\tau)/t]\tau} \tag{5.46h}$$

$$\frac{\grave{e} = \mathtt{cetpl}[L](\{i \hookrightarrow \grave{e}_i\}_{i \in L}) \qquad e = \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) \qquad \{\Omega \vdash^{\mathbb{E}} \grave{e}_i \rightsquigarrow e_i \Rightarrow \tau_i\}_{i \in L}}{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{5.46i}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \mathtt{prod}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)}{\Omega \vdash^{\mathbb{E}} \mathtt{cepr}[\ell](\grave{e}) \rightsquigarrow \mathtt{pr}[\ell](e) \Rightarrow \tau} \tag{5.46j}$$

$$\frac{n > 0 \qquad \Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \{\Omega \vdash^{\mathbb{E}} \grave{r}_i \rightsquigarrow r_i \Rightarrow \tau \Mapsto \tau'\}_{1 \leq i \leq n}}{\Omega \vdash^{\mathbb{E}} \mathtt{cematch}[n](\grave{e}; \{\grave{r}_i\}_{1 \leq i \leq n}) \rightsquigarrow \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \leq i \leq n}) \Rightarrow \tau'} \tag{5.46k}$$

$$\frac{}{\Omega, X : \mathtt{sig}\{\kappa\}(u.\tau) \vdash^{\mathbb{E}} \mathtt{ceval}(X) \rightsquigarrow \mathtt{val}(X) \Rightarrow [\mathtt{con}(X)/u]\tau} \tag{5.46l}$$

$$\frac{\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e} \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega_{\mathrm{app}} \rangle \qquad \mathrm{dom}(\Omega) \cap \mathrm{dom}(\Omega_{\mathrm{app}}) = \emptyset}{\Omega \vdash^{\hat{\Omega};\hat{\Psi};\hat{\Phi};b} \mathtt{cesplicede}[m; n] \rightsquigarrow e \Rightarrow \tau} \tag{5.46m}$$

Rules (7.6a) through (7.6k) are analagous to Rules (7.1a) through (7.1k). Rule (7.6l) governs references to spliced unexpanded expressions in synthetic position, and can be understood as described in Sec. 3.2.9.

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

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \Omega \vdash \tau <: \tau'}{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau'} \tag{5.47a}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \Omega, x : \tau \vdash^{\mathbb{E}} \grave{e}' \rightsquigarrow e' \Leftarrow \tau'}{\Omega \vdash^{\mathbb{E}} \mathtt{celetval}(\grave{e}; x.\grave{e}') \rightsquigarrow \mathtt{ap}(\mathtt{lam}\{\tau\}(x.e'); e) \Leftarrow \tau'} \tag{5.47b}$$

$$\frac{\Gamma, x : \tau_1 \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau_2}{\Omega \vdash^{\mathbb{E}} \mathtt{ceanalam}(x.\hat{e}) \rightsquigarrow \mathtt{lam}\{\tau_1\}(x.e) \Leftarrow \mathtt{parr}(\tau_1; \tau_2)} \tag{5.47c}$$

$$\frac{\Omega \vdash^{\mathbb{C}} \grave{\kappa} \rightsquigarrow \kappa \; \mathsf{kind} \qquad \Omega, u :: \kappa \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau}{\Omega \vdash^{\mathbb{E}} \mathtt{ceclam}\{\grave{\kappa}\}(u.\grave{e}) \rightsquigarrow \mathtt{clam}\{\kappa\}(u.e) \Leftarrow \mathtt{all}\{\kappa\}(u.\tau)} \tag{5.47d}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow [\mathtt{rec}(t.\tau)/t]\tau}{\Omega \vdash^{\mathbb{E}} \mathtt{cefold}(\grave{e}) \rightsquigarrow \mathtt{fold}\{t.\tau\}(e) \Leftarrow \mathtt{rec}(t.\tau)} \tag{5.47e}$$

$$\frac{\grave{e} = \mathtt{cetpl}[L](\{i \hookrightarrow \grave{e}_i\}_{i \in L}) \qquad e = \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) \qquad \{\Omega \vdash^{\mathbb{E}} \grave{e}_i \rightsquigarrow e_i \Leftarrow \tau_i\}_{i \in L}}{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{5.47f}$$

$$\frac{\grave{e} = \mathtt{cein}[\ell](\grave{e}') \qquad e = \mathtt{in}[L, \ell][\ell]\{\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau\}(e') \qquad \Delta\, \Gamma \vdash^{\mathbb{E}} \grave{e}' \rightsquigarrow e' \Leftarrow \tau}{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)} \tag{5.47g}$$

$$\frac{\Omega \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \{\Omega \vdash^{\mathbb{E}} \grave{r}_i \rightsquigarrow r_i \Leftarrow \tau \Mapsto \tau'\}_{1 \leq i \leq n}}{\Omega \vdash^{\mathbb{E}} \mathtt{cematch}[n](\grave{e}; \{\grave{r}_i\}_{1 \leq i \leq n}) \rightsquigarrow \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \leq i \leq n}) \Leftarrow \tau'} \tag{5.47h}$$

$$\frac{\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e} \qquad \hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau \qquad \hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega_{\mathrm{app}} \rangle \qquad \mathrm{dom}(\Omega) \cap \mathrm{dom}(\Omega_{\mathrm{app}}) = \emptyset}{\Omega \vdash^{\hat{\Omega}; \hat{\Psi}; \hat{\Phi}; b} \mathtt{cesplicede}[m; n] \rightsquigarrow e \Leftarrow \tau} \tag{5.47i}$$

Rules (7.7a) through (7.7h) are analagous to Rules (7.2a) through (7.2h). Rule (7.7i) 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.

$$\frac{\Omega \vdash p : \tau \dashv \Omega' \qquad \Omega \cup \Omega' \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau'}{\Omega \vdash^{\mathbb{E}} \mathtt{cerule}(p.\grave{e}) \rightsquigarrow \mathtt{rule}(p.e) \Rightarrow \tau \Mapsto \tau'} \tag{5.48}$$

The *analytic ce-rule validation judgement* is defined mutually inductively with Rules (7.2) by the following rule.

$$\frac{\Delta \vdash p : \tau \dashv \Omega' \qquad \Omega \cup \Omega' \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau'}{\Omega \vdash^{\mathbb{E}} \mathtt{cerule}(p.\grave{e}) \rightsquigarrow \mathtt{rule}(p.e) \Leftarrow \tau \Mapsto \tau'} \tag{5.49}$$

Candidate Expansion Pattern Validation

The *ce-pattern validation judgement* is inductively defined by the following rules, which are written identically to Rules (4.9).

$$\frac{}{\Omega_{\text{param}} \vdash^{\mathbb{P}} \texttt{cewildp} \rightsquigarrow \texttt{wildp} : \tau \dashv\!\mid\!\mid^{\langle \emptyset;\emptyset;\emptyset;\emptyset \rangle}} \tag{5.50a}$$

$$\frac{\Omega_{\text{param}} \vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : [\texttt{rec}(t.\tau)/t]\tau \dashv\!\mid\!\mid^{\hat{\Omega}}}{\Omega_{\text{param}} \vdash^{\mathbb{P}} \texttt{cefoldp}(\grave{p}) \rightsquigarrow \texttt{foldp}(p) : \texttt{rec}(t.\tau) \dashv\!\mid\!\mid^{\hat{\Omega}}} \tag{5.50b}$$

$$\frac{\grave{p} = \texttt{cetplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i \in L}) \qquad p = \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) \qquad \{\Omega_{\text{param}} \vdash^{\mathbb{P}} \grave{p}_i \rightsquigarrow p_i : \tau_i \dashv\!\mid\!\mid^{\hat{\Upsilon}_i}\}_{i \in L}}{\Omega_{\text{param}} \vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\!\mid\!\mid^{\cup_{i \in L} \hat{\Omega}_i}} \tag{5.50c}$$

$$\frac{\Omega_{\text{param}} \vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : \tau \dashv\!\mid\!\mid^{\hat{\Omega}}}{\Omega_{\text{param}} \vdash^{\mathbb{P}} \texttt{ceinp}[\ell](\grave{p}) \rightsquigarrow \texttt{inp}[\ell](p) : \texttt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\!\mid\!\mid^{\hat{\Omega}}} \tag{5.50d}$$

$$\frac{\mathsf{parseUPat}(\mathsf{subseq}(b;m;n)) = \hat{p} \qquad \hat{\Omega} \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid\!\mid \hat{\Omega}'}{\Omega_{\text{param}} \vdash^{\hat{\Omega};\hat{\Phi};b} \texttt{cesplicedp}[m;n] \rightsquigarrow p : \tau \dashv\!\mid\!\mid^{\hat{\Omega}'}} \tag{5.50e}$$

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 $\text{miniVerse}^{\mathbf{B}}_{\mathbf{U}}$

To formalize TSM implicits, we will now develop a reduced calculus called $\text{miniVerse}^{\mathbf{B}}_{\mathbf{U}}$, or "Bidirectional $\text{miniVerse}_{\mathbf{U}}$" (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 $\text{miniVerse}^{\mathbf{B}}_{\mathbf{U}}$ is the same as the inner core of $\text{miniVerse}_{\mathbf{U}}$, 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 $\text{miniVerse}^{\mathbf{B}}_{\mathbf{U}}$ 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 $\text{miniVerse}^{\mathbf{B}}_{\mathbf{U}}$, is defined in Figure 4.3. The full definition of the textual syntax of $\text{miniVerse}^{\mathbf{B}}_{\mathbf{U}}$ 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* $\mathsf{parseUPat}(b) = \hat{p}$.

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
			$\texttt{upr}[\ell](\hat{e})$	$\hat{e} \cdot \ell$	projection
			$\texttt{uin}[\ell](\hat{e})$	$\texttt{inj}\ \ell \cdot \hat{e}$	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
			$\texttt{uinp}[\ell](\hat{p})$	$\texttt{inj}\ \ell \cdot \hat{p}$	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 $\mathsf{miniVerse}^{\mathbf{B}}_{\mathbf{U}}$.

As in miniVerse$_{\mathbf{U}}$, 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 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. 3.2.4.
- 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:

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

  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:

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

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
$\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \ \mathsf{type}$	$\hat{\tau}$ 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 $\hat{\Gamma}$
$\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\mid \hat{Y}$	$\hat{p}$ has expansion p and type τ and generates hypotheses $\hat{Y}$ under upTSM context $\hat{\Phi}$ assuming Δ

Type Expansion

Unexpanded type formation contexts, $\hat{\Delta}$, were defined in Sec. 3.21. The *type expansion judgement*, $\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \ \mathsf{type}$, is inductively defined by Rules (3.3).

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 (7.2), respectively, as follows.

Type Synthesis *Unexpanded typing contexts*, $\hat{\Gamma}$, were defined in Sec. 3.2.5. 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.

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

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.

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

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

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

ueTSMs can be defined and applied in synthetic position.

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \text{ type} \qquad \emptyset\,\emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResultExp)} \qquad \hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uetsm}(\tau;e_{\text{parse}});\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau'}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for expressions}\ \{e_{\text{parse}}\}\ \texttt{in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \tag{7.1l}$$

$$\frac{b \downarrow e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj Success} \cdot e_{\text{cand}} \qquad e_{\text{cand}} \uparrow_{\texttt{CEExp}} \grave{e} \qquad \emptyset\,\emptyset \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uetsm}(\tau;e_{\text{parse}});\hat{\Phi};b} \grave{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uetsm}(\tau;e_{\text{parse}});\hat{\Phi}} \hat{a}\ /b/ \rightsquigarrow e \Rightarrow \tau} \tag{7.1m}$$

These rules are nearly identical to Rules (4.4l) and (4.4m), 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 3.2.6 and 3.2.7. 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 $\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}})$ when $\hat{\Psi} = \langle \mathcal{A}; \Psi; \mathcal{I} \rangle$ as shorthand for

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

TSM designation contexts, $\mathcal{I}$, are finite functions that map each type $\tau \in \text{dom}(\mathcal{I})$ 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:

$$\frac{\hat{\Delta}\,\hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle; \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau'}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}}); \mathcal{I} \rangle; \hat{\Phi}} \texttt{implicit syntax}\ \hat{a}\ \texttt{for expressions in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \tag{7.1n}$$

Like ueTSMs, upTSMs can be defined in synthetic position.

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \text{ type} \qquad \emptyset\,\emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResultPat)} \qquad \hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uptsm}(\tau;e_{\text{parse}})} \hat{e} \rightsquigarrow e \Rightarrow \tau'}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for patterns}\ \{e_{\text{parse}}\}\ \texttt{in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \tag{7.1o}$$

This rule is nearly identical to Rule (4.4n), 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 $\hat{\Phi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{uptsm}(\tau; e_{\text{parse}})$ when $\hat{\Phi} = \langle \mathcal{A}; \Phi; \mathcal{I} \rangle$ as shorthand for

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

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:

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{uptsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle} \hat{e} \rightsquigarrow e \Rightarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}}); \mathcal{I} \rangle} \texttt{implicit syntax}\ \hat{a}\ \texttt{for patterns in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \tag{7.1p}$$

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.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau} \tag{7.2a}$$

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.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e}' \rightsquigarrow e' \Leftarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{uletval}(\hat{e}; \hat{x}.\hat{e}') \rightsquigarrow \texttt{ap}(\texttt{lam}\{\tau\}(x.e'); e) \Leftarrow \tau'} \tag{7.2b}$$

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.

$$\frac{\hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau_1 \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau_2}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{uanalam}(\hat{x}.\hat{e}) \rightsquigarrow \texttt{lam}\{\tau_1\}(x.e) \Leftarrow \texttt{parr}(\tau_1; \tau_2)} \tag{7.2c}$$

Rule (7.1f) governed type lambdas in synthetic position. The following rule governs type lambdas in analytic position.

$$\frac{\hat{\Delta}, \hat{t} \rightsquigarrow t\ \text{type}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{utlam}(\hat{t}.\hat{e}) \rightsquigarrow \texttt{tlam}(t.e) \Leftarrow \texttt{all}(t.\tau)} \tag{7.2d}$$

Values of recursive types can be introduced only in analytic position.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow [\texttt{rec}(t.\tau)/t]\tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{ufold}(\hat{e}) \rightsquigarrow \texttt{fold}\{t.\tau\}(e) \Leftarrow \texttt{rec}(t.\tau)} \tag{7.2e}$$

Rule (7.1i) governed labeled tuples in synthetic position. The following rule governs labeled tuples in analytic position.

$$\frac{\{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_i \rightsquigarrow e_i \Leftarrow \tau_i\}_{i \in L}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{utpl}[L](\{i \hookrightarrow \hat{e}_i\}_{i \in L}) \rightsquigarrow \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) \Leftarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{7.2f}$$

Values of labeled sum type can appear only in analytic position.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\left(\begin{array}{c}\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uin}[\ell](\hat{e}) \\ \rightsquigarrow \\ \mathtt{in}[L,\ell][\ell]\{\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau\}(e) \Leftarrow \mathtt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)\end{array}\right)} \tag{7.2g}$$

Rule (7.1k) governed match expressions in synthetic position. The following rule governs match expressions in analytic position.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \rightsquigarrow r_i \Leftarrow \tau \Rightarrow \tau'\}_{1 \leq i \leq n}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{umatch}[n](\hat{e}; \{\hat{r}_i\}_{1 \leq i \leq n}) \rightsquigarrow \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \leq i \leq n}) \Leftarrow \tau'} \tag{7.2h}$$

Rule (7.1l) governed ueTSM definitions in synthetic position. The following rule governs ueTSM definitions in analytic position.

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \varnothing\ \varnothing \vdash e_{\mathrm{parse}} : \mathtt{parr}(\mathtt{Body}; \mathtt{ParseResultExp}) \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi},\hat{a} \rightsquigarrow a \hookrightarrow \mathtt{uetsm}(\tau; e_{\mathrm{parse}});\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{for\ expressions}\ \{e_{\mathrm{parse}}\}\ \mathtt{in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{7.2i}$$

Rule (7.1n) governed ueTSM designations in synthetic position. The following rule governs ueTSM designations in analytic position.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \mathtt{uetsm}(\tau; e_{\mathrm{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle;\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \mathtt{uetsm}(\tau; e_{\mathrm{parse}}); \mathcal{I} \rangle;\hat{\Phi}} \mathtt{implicit\ syntax}\ \hat{a}\ \mathtt{for\ expressions\ in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{7.2j}$$

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 (7.1m).

$$\frac{b \downarrow e_{\mathrm{body}} \qquad e_{\mathrm{parse}}(e_{\mathrm{body}}) \Downarrow \mathtt{inj}\ \mathtt{Success} \cdot e_{\mathrm{cand}} \qquad e_{\mathrm{cand}} \uparrow_{\mathsf{CEExp}} \grave{e} \qquad \varnothing\ \varnothing \vdash^{\hat{\Delta};\hat{\Gamma};\langle \mathcal{A};\Psi,a \hookrightarrow \mathtt{uetsm}(\tau;e_{\mathrm{parse}});\mathcal{I} \uplus \tau \hookrightarrow a \rangle;\hat{\Phi};b} \grave{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A};\Psi,a \hookrightarrow \mathtt{uetsm}(\tau;e_{\mathrm{parse}});\mathcal{I} \uplus \tau \hookrightarrow a \rangle;\hat{\Phi}} \mathtt{uelit}[b] \rightsquigarrow e \Leftarrow \tau} \tag{7.2k}$$

Rule (7.1o) governed upTSM definitions in synthetic position. The following rule governs upTSM definitions in analytic position.

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \varnothing\ \varnothing \vdash e_{\mathrm{parse}} : \mathtt{parr}(\mathtt{Body}; \mathtt{ParseResultPat}) \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi},\hat{\Phi},\hat{a} \rightsquigarrow a \hookrightarrow \mathtt{uptsm}(\tau; e_{\mathrm{parse}})} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{for\ patterns}\ \{e_{\mathrm{parse}}\}\ \mathtt{in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{7.2l}$$

Rule (7.1p) governed upTSM designations in synthetic position. The following rule governs upTSM designations in analytic position.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \mathtt{uptsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \mathtt{uetsm}(\tau; e_{\text{parse}}); \mathcal{I} \rangle} \mathtt{implicit\ syntax}\ \hat{a}\ \mathtt{for\ patterns\ in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{7.2m}$$

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 (7.2) by the following rule.

$$\frac{\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \langle \mathcal{G}'; \Gamma' \rangle \qquad \langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G} \uplus \mathcal{G}'; \Gamma \cup \Gamma' \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau'}{\langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{urule}(\hat{p}.\hat{e}) \rightsquigarrow \mathtt{rule}(p.e) \Rightarrow \tau \Mapsto \tau'} \tag{7.3}$$

The analytic typed rule expansion judgement is invoked iteratively by Rule (7.2h). This judgement is defined mutually inductively with Rules (7.1), Rules (7.2), and Rule (7.3) by the following rule, which is the analytic analag of Rule (7.3).

$$\frac{\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \langle \mathcal{G}'; \Gamma' \rangle \qquad \langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G} \uplus \mathcal{G}'; \Gamma \cup \Gamma' \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{urule}(\hat{p}.\hat{e}) \rightsquigarrow \mathtt{rule}(p.e) \Leftarrow \tau \Mapsto \tau'} \tag{7.4}$$

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 (7.5) as follows.

The following rules are written identically to the typed pattern expansion rules for shared pattern forms in miniVerse$_{\mathsf{U}}$, i.e. Rules (4.6a) through (4.6e).

$$\frac{}{\Delta \vdash_{\hat{\Phi}} \hat{x} \rightsquigarrow x : \tau \dashv\!\mid \langle \hat{x} \rightsquigarrow x; x : \tau \rangle} \tag{7.5a}$$

$$\frac{}{\Delta \vdash_{\hat{\Phi}} \mathtt{uwildp} \rightsquigarrow \mathtt{wildp} : \tau \dashv\!\mid \langle \emptyset; \emptyset \rangle} \tag{7.5b}$$

$$\frac{\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : [\mathtt{rec}(t.\tau)/t]\tau \dashv\!\mid \hat{Y}}{\Delta \vdash_{\hat{\Phi}} \mathtt{ufoldp}(\hat{p}) \rightsquigarrow \mathtt{foldp}(p) : \mathtt{rec}(t.\tau) \dashv\!\mid \hat{Y}} \tag{7.5c}$$

$$\frac{\{\Delta \vdash_{\hat{\Phi}} \hat{p}_i \rightsquigarrow p_i : \tau_i \dashv\!\mid \hat{Y}_i\}_{i \in L}}{\left(\begin{array}{c} \Delta \vdash_{\hat{\Phi}} \mathtt{utplp}[L](\{i \hookrightarrow \hat{p}_i\}_{i \in L}) \\ \rightsquigarrow \\ \mathtt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\!\mid \cup_{i \in L} \hat{Y}_i \end{array}\right)} \tag{7.5d}$$

$$\frac{\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \hat{Y}}{\Delta \vdash_{\hat{\Phi}} \mathtt{uinp}[\ell](\hat{p}) \rightsquigarrow \mathtt{inp}[\ell](p) : \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\!\mid \hat{Y}} \tag{7.5e}$$

The following rule governs upTSM application. It is written identically to Rule (4.6f).

$$\frac{\begin{array}{c} b \downarrow e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}\,\texttt{Success} \cdot e_{\text{cand}} \qquad e_{\text{cand}} \uparrow_{\mathsf{CEPat}} \grave{p} \\ \vdash^{\Delta;\hat{\Phi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uptsm}(\tau;e_{\text{parse}});b} \grave{p} \rightsquigarrow p : \tau \dashv\!\!\mid^{\hat{Y}} \end{array}}{\Delta \vdash_{\hat{\Phi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uptsm}(\tau;e_{\text{parse}})} \hat{a}\ /b/ \rightsquigarrow p : \tau \dashv\!\!\mid \hat{Y}} \tag{7.5f}$$

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 (7.5f).

$$\frac{\begin{array}{c} b \downarrow e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}\,\texttt{Success} \cdot e_{\text{cand}} \qquad e_{\text{cand}} \uparrow_{\mathsf{CEPat}} \grave{p} \\ \vdash^{\Delta;\langle \mathcal{A};\Phi,a\hookrightarrow \texttt{uptsm}(\tau;e_{\text{parse}});\mathcal{I},\tau\hookrightarrow a\rangle;b} \grave{p} \rightsquigarrow p : \tau \dashv\!\!\mid^{\hat{Y}} \end{array}}{\Delta \vdash_{\langle \mathcal{A};\Phi,a\hookrightarrow \texttt{uptsm}(\tau;e_{\text{parse}});\mathcal{I},\tau\hookrightarrow a\rangle} /b/ \rightsquigarrow p : \tau \dashv\!\!\mid \hat{Y}} \tag{7.5g}$$

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 $\mathcal{U}(Y)$ was defined in 4.2.5.

Lemma 7.2 (Pattern Expressibility)**.** *If* $\Delta \vdash p : \tau \dashv\!\!\mid Y$ *then* $\Delta \vdash_{\hat{\Phi}} \mathcal{U}(p) \rightsquigarrow p : \tau \dashv\!\!\mid \mathcal{U}(Y)$.

Proof. By rule induction over Rules (4.3), using the definitions of $\mathcal{U}(Y)$ and $\mathcal{U}(p)$. In each case, we can apply the IH to or over each premise, then apply the corresponding rule in Rules (7.5). □

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 (4.1) and Rule (4.2) 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 3.13 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 (7.3). □

Sort			Operational Form	Stylized Form	Description
CETyp	$\grave{\tau}$	::=	t	t	variable
			$\texttt{ceparr}(\grave{\tau}; \grave{\tau})$	$\grave{\tau} \rightharpoonup \grave{\tau}$	partial function
			$\texttt{ceall}(t.\grave{\tau})$	$\forall t.\grave{\tau}$	polymorphic
			$\texttt{cerec}(t.\grave{\tau})$	$\mu t.\grave{\tau}$	recursive
			$\texttt{ceprod}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L})$	$\langle \{i \hookrightarrow \grave{\tau}_i\}_{i \in L} \rangle$	labeled product
			$\texttt{cesum}[L](\{i \hookrightarrow \grave{\tau}_i\}_{i \in L})$	$[\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}]$	labeled sum
			$\texttt{cesplicedt}[m; n]$	$\texttt{splicedt}\langle m, n \rangle$	spliced
CEExp	$\grave{e}$	::=	x	x	variable
			$\texttt{ceasc}\{\grave{\tau}\}(\grave{e})$	$\grave{e} : \grave{\tau}$	ascription
			$\texttt{celetval}(\grave{e}; x.\grave{e})$	$\texttt{let val}\ x = \grave{e}\ \texttt{in}\ \grave{e}$	value binding
			$\texttt{ceanalam}(x.\grave{e})$	$\lambda x.\grave{e}$	abstraction (unannotated)
			$\texttt{celam}\{\grave{\tau}\}(x.\grave{e})$	$\lambda x{:}\grave{\tau}.\grave{e}$	abstraction (annotated)
			$\texttt{ceap}(\grave{e}; \grave{e})$	$\grave{e}(\grave{e})$	application
			$\texttt{cetlam}(t.\grave{e})$	$\Lambda t.\grave{e}$	type abstraction
			$\texttt{cetap}\{\grave{\tau}\}(\grave{e})$	$\grave{e}[\grave{\tau}]$	type application
			$\texttt{cefold}(\grave{e})$	$\texttt{fold}(\grave{e})$	fold
			$\texttt{ceunfold}(\grave{e})$	$\texttt{unfold}(\grave{e})$	unfold
			$\texttt{cetpl}[L](\{i \hookrightarrow \grave{e}_i\}_{i \in L})$	$\langle \{i \hookrightarrow \grave{e}_i\}_{i \in L} \rangle$	labeled tuple
			$\texttt{cepr}[\ell](\grave{e})$	$\grave{e} \cdot \ell$	projection
			$\texttt{cein}[\ell](\grave{e})$	$\texttt{inj}\ \ell \cdot \grave{e}$	injection
			$\texttt{cematch}[n](\grave{e}; \{\grave{r}_i\}_{1 \leq i \leq n})$	$\texttt{match}\ \grave{e}\ \{\grave{r}_i\}_{1 \leq i \leq n}$	match
			$\texttt{cesplicede}[m; n]$	$\texttt{splicede}\langle m, n \rangle$	spliced
CERule	$\grave{r}$	::=	$\texttt{cerule}(p.\grave{e})$	$p \Rightarrow \grave{e}$	rule
CEPat	$\grave{p}$	::=	$\texttt{cewildp}$	$\_$	wildcard pattern
			$\texttt{cefoldp}(p)$	$\texttt{fold}(p)$	fold pattern
			$\texttt{cetplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i \in L})$	$\langle \{i \hookrightarrow \grave{p}_i\}_{i \in L} \rangle$	labeled tuple pattern
			$\texttt{ceinp}[\ell](\grave{p})$	$\texttt{inj}\ \ell \cdot \grave{p}$	injection pattern
			$\texttt{cesplicedp}[m; n]$	$\texttt{splicedp}\langle m, n \rangle$	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 $\mathcal{C}(\tau)$, defined in Sec. 3.2.8.
- Each expanded expression form maps onto a ce-expression form according to the metafunction $\mathcal{C}(e)$, defined as follows:

$$
\begin{aligned}
\mathcal{C}(x) &= x \\
\mathcal{C}(\texttt{lam\{}\tau\texttt{\}(}x.e\texttt{)}) &= \texttt{celam\{}\mathcal{C}(\tau)\texttt{\}(}x.\mathcal{C}(e)\texttt{)} \\
\mathcal{C}(\texttt{ap(}e_1;e_2\texttt{)}) &= \texttt{ceap(}\mathcal{C}(e_1);\mathcal{C}(e_2)\texttt{)} \\
\mathcal{C}(\texttt{tlam(}t.e\texttt{)}) &= \texttt{cetlam(}t.\mathcal{C}(e)\texttt{)} \\
\mathcal{C}(\texttt{tap\{}\tau\texttt{\}(}e\texttt{)}) &= \texttt{cetap\{}\mathcal{C}(\tau)\texttt{\}(}\mathcal{C}(e)\texttt{)} \\
\mathcal{C}(\texttt{fold\{}t.\tau\texttt{\}(}e\texttt{)}) &= \texttt{ceasc\{cerec(}t.\mathcal{C}(\tau)\texttt{)\}(cefold(}\mathcal{C}(e)\texttt{))} \\
\mathcal{C}(\texttt{unfold(}e\texttt{)}) &= \texttt{ceunfold(}\mathcal{C}(e)\texttt{)} \\
\mathcal{C}(\texttt{tpl}[L]\texttt{(\{}i \hookrightarrow e_i\texttt{\}}_{i\in L}\texttt{)}) &= \texttt{cetpl}[L]\texttt{(\{}i \hookrightarrow \mathcal{C}(e_i)\texttt{\}}_{i\in L}\texttt{)} \\
\mathcal{C}(\texttt{in}[L]\texttt{[}\ell\texttt{]\{\{}i \hookrightarrow \tau_i\texttt{\}}_{i\in L}\texttt{\}(}e\texttt{)}) &= \texttt{ceasc\{cesum}[L]\texttt{(\{}i \hookrightarrow \mathcal{C}(\tau_i)\texttt{\}}_{i\in L}\texttt{)\}(cein[}\ell\texttt{](}\mathcal{C}(e)\texttt{))} \\
\mathcal{C}(\texttt{match}[n]\texttt{\{}\tau\texttt{\}(}e;\{r_i\}_{1\le i\le n}\texttt{)}) &= \texttt{ceasc\{}\mathcal{C}(\tau)\texttt{\}(cematch}[n]\texttt{(}\mathcal{C}(e);\{\mathcal{C}(r_i)\}_{1\le i\le n}\texttt{))}
\end{aligned}
$$

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

$$\mathcal{C}(\texttt{rule(}p.e\texttt{)}) = \texttt{cerule(}p.\mathcal{C}(e)\texttt{)}$$

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

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

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 $\mathbb{E}$
$\vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : \tau \dashv\mid^{\hat{Y}}$	$\grave{p}$ expands to p and matches values of type τ generating assumptions $\hat{Y}$ 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 (4.4m) and (4.6f)), 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 (3.5), which were defined in Sec. 3.2.9.

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 (7.2) by Rules (7.6) and Rules (7.7), respectively, as follows.

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

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash^{\mathbb{E}} x \rightsquigarrow x \Rightarrow \tau} \tag{7.6a}$$

$$\frac{\Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau} \rightsquigarrow \tau \; \mathsf{type} \qquad \Delta\; \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau}{\Delta\; \Gamma \vdash^{\mathbb{E}} \mathtt{ceasc\{\grave{\tau}\}(\grave{e})} \rightsquigarrow e \Rightarrow \tau} \tag{7.6b}$$

$$\frac{\Delta\; \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \Delta\; \Gamma, x : \tau \vdash^{\mathbb{E}} \grave{e}' \rightsquigarrow e' \Rightarrow \tau'}{\Delta\; \Gamma \vdash^{\mathbb{E}} \mathtt{celetval}(\grave{e}; x.\grave{e}') \rightsquigarrow \mathtt{ap}(\mathtt{lam}\{\tau\}(x.e'); e) \Rightarrow \tau'} \tag{7.6c}$$

$$\frac{\Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau}_1 \rightsquigarrow \tau_1 \; \mathsf{type} \qquad \Delta\; \Gamma, x : \tau_1 \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau_2}{\Delta\; \Gamma \vdash^{\mathbb{E}} \mathtt{celam}\{\grave{\tau}_1\}(x.\grave{e}) \rightsquigarrow \mathtt{lam}\{\tau_1\}(x.e) \Rightarrow \mathtt{parr}(\tau_1; \tau_2)} \tag{7.6d}$$

$$\frac{\Delta\; \Gamma \vdash^{\mathbb{E}} \grave{e}_1 \rightsquigarrow e_1 \Rightarrow \mathtt{parr}(\tau_2; \tau) \qquad \Delta\; \Gamma \vdash^{\mathbb{E}} \grave{e}_2 \rightsquigarrow e_2 \Leftarrow \tau_2}{\Delta\; \Gamma \vdash^{\mathbb{E}} \mathtt{ceap}(\grave{e}_1; \grave{e}_2) \rightsquigarrow \mathtt{ap}(e_1; e_2) \Rightarrow \tau} \tag{7.6e}$$

$$\frac{\Delta, t \; \mathsf{type}\; \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau}{\Delta\; \Gamma \vdash^{\mathbb{E}} \mathtt{cetlam}(t.\grave{e}) \rightsquigarrow \mathtt{tlam}(t.e) \Rightarrow \mathtt{all}(t.\tau)} \tag{7.6f}$$

$$\frac{\Delta\; \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \mathtt{all}(t.\tau) \qquad \Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau}' \rightsquigarrow \tau' \; \mathsf{type}}{\Delta\; \Gamma \vdash^{\mathbb{E}} \mathtt{cetap}\{\grave{\tau}'\}(\grave{e}) \rightsquigarrow \mathtt{tap}\{\tau'\}(e) \Rightarrow [\tau'/t]\tau} \tag{7.6g}$$

$$\frac{\Delta\; \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \mathtt{rec}(t.\tau)}{\Delta\; \Gamma \vdash^{\mathbb{E}} \mathtt{ceunfold}(\grave{e}) \rightsquigarrow \mathtt{unfold}(e) \Rightarrow [\mathtt{rec}(t.\tau)/t]\tau} \tag{7.6h}$$

$$\frac{\{\Delta\; \Gamma \vdash^{\mathbb{E}} \grave{e}_i \rightsquigarrow e_i \Rightarrow \tau_i\}_{i \in L}}{\Delta\; \Gamma \vdash^{\mathbb{E}} \mathtt{cetpl}[L](\{i \hookrightarrow \grave{e}_i\}_{i \in L}) \rightsquigarrow \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) \Rightarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{7.6i}$$

$$\frac{\Delta\; \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \mathtt{prod}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)}{\Delta\; \Gamma \vdash^{\mathbb{E}} \mathtt{cepr}[\ell](\grave{e}) \rightsquigarrow \mathtt{pr}[\ell](e) \Rightarrow \tau} \tag{7.6j}$$

$$\frac{n > 0 \qquad \Delta\; \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \{\Delta\; \Gamma \vdash^{\mathbb{E}} \grave{r}_i \rightsquigarrow r_i \Rightarrow \tau \Rightarrow \tau'\}_{1 \leq i \leq n}}{\Delta\; \Gamma \vdash^{\mathbb{E}} \mathtt{cematch}[n](\grave{e}; \{\grave{r}_i\}_{1 \leq i \leq n}) \rightsquigarrow \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \leq i \leq n}) \Rightarrow \tau'} \tag{7.6k}$$

$$\frac{\begin{array}{c}\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e} \qquad \langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle \; \langle \mathcal{G}; \Gamma_{\mathsf{app}} \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \\ \Delta \cap \Delta_{\mathsf{app}} = \emptyset \qquad \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma_{\mathsf{app}}) = \emptyset\end{array}}{\Delta\; \Gamma \vdash^{\langle \mathcal{D}; \Delta_{\mathsf{app}} \rangle; \langle \mathcal{G}; \Gamma_{\mathsf{app}} \rangle; \hat{\Psi}; \hat{\Phi}; b} \mathtt{cesplicede}[m; n] \rightsquigarrow e \Rightarrow \tau} \tag{7.6l}$$

Rules (7.6a) through (7.6k) are analagous to Rules (7.1a) through (7.1k). Rule (7.6l) governs references to spliced unexpanded expressions in synthetic position, and can be understood as described in Sec. 3.2.9.

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

$$\frac{\Delta\; \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau}{\Delta\; \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau} \tag{7.7a}$$

$$\frac{\Delta\; \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \Delta\; \Gamma, x : \tau \vdash^{\mathbb{E}} \grave{e}' \rightsquigarrow e' \Leftarrow \tau'}{\Delta\; \Gamma \vdash^{\mathbb{E}} \mathtt{celetval}(\grave{e}; x.\grave{e}') \rightsquigarrow \mathtt{ap}(\mathtt{lam}\{\tau\}(x.e'); e) \Leftarrow \tau'} \tag{7.7b}$$

$$\frac{\Delta\ \Gamma, x : \tau_1 \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau_2}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{ceanalam}(x.\hat{e}) \rightsquigarrow \mathtt{lam}\{\tau_1\}(x.e) \Leftarrow \mathtt{parr}(\tau_1;\tau_2)} \tag{7.7c}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{cetlam}(t.\grave{e}) \rightsquigarrow \mathtt{tlam}(t.e) \Leftarrow \mathtt{all}(t.\tau)} \tag{7.7d}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow [\mathtt{rec}(t.\tau)/t]\tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{cefold}(\grave{e}) \rightsquigarrow \mathtt{fold}\{t.\tau\}(e) \Leftarrow \mathtt{rec}(t.\tau)} \tag{7.7e}$$

$$\frac{\{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e}_i \rightsquigarrow e_i \Leftarrow \tau_i\}_{i \in L}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{cetpl}[L](\{i \hookrightarrow \grave{e}_i\}_{i \in L}) \rightsquigarrow \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) \Leftarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{7.7f}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau}{\left(\begin{array}{c}\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{cein}[\ell](\grave{e}) \\ \overset{\rightsquigarrow}{\mathtt{in}[L,\ell][\ell]\{\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau\}(e)} \Leftarrow \mathtt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)\end{array}\right)} \tag{7.7g}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau \qquad \{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r}_i \rightsquigarrow r_i \Leftarrow \tau \Mapsto \tau'\}_{1 \leq i \leq n}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{cematch}[n](\grave{e}; \{\grave{r}_i\}_{1 \leq i \leq n}) \rightsquigarrow \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \leq i \leq n}) \Leftarrow \tau'} \tag{7.7h}$$

$$\frac{\begin{array}{c}\mathsf{parseUExp}(\mathsf{subseq}(b;m;n)) = \hat{e} \qquad \langle \mathcal{D}; \Delta_{\mathrm{app}} \rangle\ \langle \mathcal{G}; \Gamma_{\mathrm{app}} \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau \\ \Delta \cap \Delta_{\mathrm{app}} = \emptyset \qquad \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma_{\mathrm{app}}) = \emptyset\end{array}}{\Delta\ \Gamma \vdash^{\langle \mathcal{D};\Delta_{\mathrm{app}} \rangle; \langle \mathcal{G};\Gamma_{\mathrm{app}} \rangle; \hat{\Psi}; \hat{\Phi}; b} \mathtt{cesplicede}[m;n] \rightsquigarrow e \Leftarrow \tau} \tag{7.7i}$$

Rules (7.7a) through (7.7h) are analagous to Rules (7.2a) through (7.2h). Rule (7.7i) 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.

$$\frac{\Delta \vdash p : \tau \dashv\!\mid \Upsilon \qquad \Delta\ \Gamma \cup \Upsilon \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Rightarrow \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{cerule}(p.\grave{e}) \rightsquigarrow \mathtt{rule}(p.e) \Rightarrow \tau \Mapsto \tau'} \tag{7.8}$$

The *analytic ce-rule validation judgement* is defined mutually inductively with Rules (7.2) by the following rule.

$$\frac{\Delta \vdash p : \tau \dashv\!\mid \Upsilon \qquad \Delta\ \Gamma \cup \Upsilon \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e \Leftarrow \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{cerule}(p.\grave{e}) \rightsquigarrow \mathtt{rule}(p.e) \Leftarrow \tau \Mapsto \tau'} \tag{7.9}$$

Candidate Expansion Pattern Validation

The *ce-pattern validation judgement* is inductively defined by the following rules, which are written identically to Rules (4.9).

$$\frac{}{\vdash^{\mathbb{P}} \texttt{cewildp} \rightsquigarrow \texttt{wildp} : \tau \dashv\!\mid^{\langle \emptyset;\emptyset \rangle}} \tag{7.10a}$$

$$\frac{\vdash^{\mathbb{P}} \hat{p} \rightsquigarrow p : [\texttt{rec}(t.\tau)/t]\tau \dashv\!\mid^{\hat{Y}}}{\vdash^{\mathbb{P}} \texttt{cefoldp}(\hat{p}) \rightsquigarrow \texttt{foldp}(p) : \texttt{rec}(t.\tau) \dashv\!\mid^{\hat{Y}}} \tag{7.10b}$$

$$\frac{\{\vdash^{\mathbb{P}} \hat{p}_i \rightsquigarrow p_i : \tau_i \dashv\!\mid^{\hat{Y}_i}\}_{i \in L}}{\left(\begin{array}{c} \vdash^{\mathbb{P}} \texttt{cetplp}[L](\{i \hookrightarrow \hat{p}_i\}_{i \in L}) \\ \rightsquigarrow \\ \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\!\mid^{\cup_{i \in L} \hat{Y}_i} \end{array}\right)} \tag{7.10c}$$

$$\frac{\vdash^{\mathbb{P}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid^{\hat{Y}}}{\vdash^{\mathbb{P}} \texttt{ceinp}[\ell](\hat{p}) \rightsquigarrow \texttt{inp}[\ell](p) : \texttt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\!\mid^{\hat{Y}}} \tag{7.10d}$$

$$\frac{\mathsf{parseUPat}(\mathsf{subseq}(b;m;n)) = \hat{p} \qquad \Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \hat{Y}}{\vdash^{\Delta;\hat{\Phi};b} \texttt{cesplicedp}[m;n] \rightsquigarrow p : \tau \dashv\!\mid^{\hat{Y}}} \tag{7.10e}$$

Candidate Expansion Expressibility

The following lemma establishes that each well-typed expanded expression, e, can be expressed as a valid ce-expression, $\mathcal{C}(e)$, 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* $\Delta\ \Gamma \vdash^{\mathbb{E}} \mathcal{C}(e) \rightsquigarrow e \Rightarrow \tau$.
2. *If* $\Delta\ \Gamma \vdash r : \tau \Mapsto \tau'$ *then* $\Delta\ \Gamma \vdash^{\mathbb{E}} \mathcal{C}(r) \rightsquigarrow r \Rightarrow \tau \Mapsto \tau'$.

Proof. By mutual rule induction over Rules (4.1) and Rule (4.2). 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 3.18 to or over each type formation premise and then derive the conclusion by applying Rules (7.6) and Rule (7.8) 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* $\Delta \vdash p : \tau \dashv\!\mid \emptyset$ *then* $\vdash^{\Delta;\hat{\Phi};b} \mathcal{C}(p) \rightsquigarrow p : \tau \dashv\!\mid^{\langle \emptyset;\emptyset \rangle}$.

Proof. The proof is nearly identical to the proof of Lemma 4.20, differing only in that each mention of a rule in Rules (4.9) is replaced by a mention of the corresponding rule in Rules (7.10). □

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* $\Delta \vdash_{\langle \mathcal{A};\Phi;\mathcal{I}\rangle} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \langle \mathcal{G};\mathcal{Y}\rangle$ *then* $\Delta \vdash p : \tau \dashv\!\mid \mathcal{Y}$.
2. *If* $\vdash^{\Delta;\langle \mathcal{A};\Phi;\mathcal{I}\rangle;b} \grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\langle \mathcal{G};\mathcal{Y}\rangle}$ *then* $\Delta \vdash p : \tau \dashv\!\mid \mathcal{Y}$.

Proof. My mutual rule induction over Rules (7.5) and Rules (7.10).

1. We induct on the premise. In the following, let $\hat{\mathcal{Y}} = \langle \mathcal{G};\mathcal{Y}\rangle$ and $\hat{\Phi} = \langle \mathcal{A};\Phi;\mathcal{I}\rangle$.
   Case (7.5a) through (7.5f)**.** In each of these cases, the proof is written identically to the proof of the corresponding case in the proof of Theorem 4.21.
   Case (7.5g)**.** We have:
   (1) $\hat{p} = \texttt{uplit}[b]$ by assumption
   (2) $\Phi = \Phi', a \hookrightarrow \texttt{uptsm}(\tau;\, e_{\text{parse}})$ by assumption
   (3) $\mathcal{I} = \mathcal{I}', \tau \hookrightarrow a$ by assumption
   (4) $b \downarrow e_{\text{body}}$ by assumption
   (5) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj Success} \cdot e_{\text{cand}}$ by assumption
   (6) $e_{\text{cand}} \uparrow_{\mathsf{CEPat}} \grave{p}$ by assumption
   (7) $\vdash^{\Delta;\langle \mathcal{A};\Phi', a\hookrightarrow\texttt{uptsm}(\tau;e_{\text{parse}});\mathcal{I}', \tau\hookrightarrow a\rangle;b} \grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\langle \mathcal{G};\mathcal{Y}\rangle}$ by assumption
   (8) $\Delta \vdash p : \tau \dashv\!\mid \mathcal{Y}$ by IH, part 2 on (7)
2. We induct on the premise. In the following, let $\hat{\mathcal{Y}} = \langle \mathcal{G};\mathcal{Y}\rangle$ and $\hat{\Phi} = \langle \mathcal{A};\Phi;\mathcal{I}\rangle$.
   Case (7.10a) through (7.10e)**.** In each case, the proof is written identically to the proof of the corresponding case in the proof of Theorem 4.21.

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:

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

where $\|b\|$ is the length of b and $\|\hat{p}\|$ is the sum of the lengths of the literal bodies in $\hat{p}$,

$$\begin{aligned}
\|\hat{x}\| &= 0 \\
\|\texttt{ufoldp}(\hat{p})\| &= \|\hat{p}\| \\
\|\texttt{utplp}[L](\{i \hookrightarrow \hat{p}_i\}_{i\in L})\| &= \sum_{i\in L} \|\hat{p}_i\| \\
\|\texttt{uinp}[\ell](\hat{p})\| &= \|\hat{p}\| \\
\|\texttt{uapuptsm}[b][\hat{a}]\| &= \|b\| \\
\|\texttt{uplit}[b]\| &= \|b\|
\end{aligned}$$

The only case in the proof of part 1 that invokes part 2 are Case (7.5f) and (7.5g). There, we have that the metric remains stable:

$$\begin{aligned}
&\|\Delta \vdash_{\hat{\Phi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uetsm}(\tau;e_{\text{parse}})} \texttt{uapuptsm}[b]\,\texttt{[}\hat{a}\texttt{]} \rightsquigarrow p : \tau \dashv\| \hat{Y}\| \\
=&\|\Delta \vdash_{\langle \mathcal{A};\Phi',a\hookrightarrow \texttt{uptsm}(\tau;e_{\text{parse}});\mathcal{I}',\tau\hookrightarrow a\rangle} \texttt{uplit}[b] \rightsquigarrow p : \tau \dashv\| \hat{Y}\| \\
=&\|\vdash^{\Delta;\hat{\Phi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uetsm}(\tau;e_{\text{parse}});b} \grave{p} \rightsquigarrow p : \tau \dashv\|^{\hat{Y}}\| \\
=&\|b\|
\end{aligned}$$

The only case in the proof of part 2 that invokes part 1 is Case (7.10e). There, we have that $\mathsf{parseUPat}(\mathsf{subseq}(b;m;n)) = \hat{p}$ and the IH is applied to the judgement $\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\| \hat{Y}$. 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:

$$\|\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\| \hat{Y}\| < \|\vdash^{\Delta;\hat{\Phi};b} \texttt{cesplicedp}[m;n] \rightsquigarrow p : \tau \dashv\|^{\hat{Y}}\|$$

i.e. by the definitions above,

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

This is established by appeal to Condition 3.22, 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 3.22 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* $\Delta \vdash \Psi$ ueTSMs *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 (7.2), Rule (7.3), Rule (7.4), Rules (7.6), Rules (7.7), Rule (7.8) and Rule (7.9). In the following, we refer to the induction hypothesis applied to the assumption $\Delta \vdash \Psi$ ueTSMs 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 (4.1a)

   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 3.12 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 3.12 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 (4.1) and Rule (4.2) as needed.

   Case (7.1l)**.** 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)	$\emptyset\ \emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResultExp)}$	by assumption
(4)	$\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a;\Psi,a \hookrightarrow \texttt{uetsm}(\tau';e_{\text{parse}});\mathcal{I}\rangle;\hat{\Phi}} \hat{e}' \rightsquigarrow e \Rightarrow \tau$	by assumption
(5)	$\Delta \vdash \Psi$ ueTSMs	by assumption
(6)	$\Delta \vdash \tau'$ type	by Lemma 3.12 to (2)

(7) $\Delta \vdash \Psi, \hat{a} \hookrightarrow \texttt{uetsm}(\tau'; e_{\text{parse}})$ ueTSMs — by Definition 4.14 on (5), (6) and (3)

(8) $\Delta\, \Gamma \vdash e : \tau$ — by Outer IH, part 1(a)(i) on (7) and (4)

Case (7.1m)**.** We have:

(1) $\hat{e} = \texttt{uapuetsm}[b][\hat{a}]$ — by assumption

(2) $\hat{\Psi} = \langle \mathcal{A}' \uplus \hat{a} \rightsquigarrow a; \Psi', a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}}); \mathcal{I} \rangle$ — by assumption

(3) $b \downarrow e_{\text{body}}$ — by assumption

(4) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj Success} \cdot e_{\text{cand}}$ — by assumption

(5) $e_{\text{cand}} \uparrow_{\mathsf{CEExp}} \grave{e}$ — by assumption

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

(7) $\Delta \vdash \Psi$ ueTSMs — by assumption

(8) $\Delta \vdash \tau$ type — by Definition 4.14 on (7)

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

(10) $\emptyset \cap \mathrm{dom}(\Gamma) = \emptyset$ — 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 (7.1n)**.** We have:

(1) $\hat{e} = \texttt{uimplicite}[\hat{a}](\hat{e})$ — by assumption

(2) $\hat{\Psi} = \langle \mathcal{A}' \uplus \hat{a} \rightsquigarrow a; \Psi', a \hookrightarrow \texttt{uetsm}(\tau'; e_{\text{parse}}); \mathcal{I} \rangle$ — by assumption

(3) $\hat{\Delta}\, \hat{\Gamma} \vdash_{\langle \mathcal{A}' \uplus \hat{a} \rightsquigarrow a; \Psi', a \hookrightarrow \texttt{uetsm}(\tau'; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle; \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau$ — by assumption

(4) $\Delta\, \Gamma \vdash e : \tau$ — by IH, part 1(a)(i) on (3)

Case (7.1o)**.** 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) $\hat{\Delta}\, \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{uptsm}(\tau'; e_{\text{parse}})} \hat{e}' \rightsquigarrow e \Rightarrow \tau$ — by assumption

(4) $\Delta\, \Gamma \vdash e : \tau$ — by IH, part 1(a)(i) on (3)

Case (7.1p)**.** We have:

(1) $\hat{e} = \texttt{uimplicitp}[\hat{a}](\hat{e})$ — by assumption

(2) $\hat{\Phi} = \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{uptsm}(\tau'; e_{\text{parse}}); \mathcal{I} \rangle$

(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \mathtt{uptsm}(\tau'; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle} \hat{e} \rightsquigarrow e \Rightarrow \tau$ 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 (7.3)**.** We have:

(1) $\hat{r} = \mathtt{urule}(\hat{p}.\hat{e})$ by assumption

(2) $r = \mathtt{rule}(p.e)$ by assumption

(3) $\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\mid \langle \mathcal{A}'; \mathcal{Y} \rangle$ by assumption

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

(5) $\Delta \vdash p : \tau \dashv\mid \mathcal{Y}$ by Theorem 7.6, part 1 on (3)

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

(7) $\Delta\ \Gamma \vdash \mathtt{rule}(p.e) : \tau \Mapsto \tau'$ by Rule (4.2) on (5) and (6)

(b) i. We induct on the assumption.

Case (7.2a)**.** 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 (7.2b) through (7.2h)**.** In each of these cases, we apply:

- Lemma 3.12 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 (4.1) and Rule (4.2) as needed.

Case (7.2i)**.** We have:

(3) $\hat{e} = \mathtt{usyntaxue}\{e_{\text{parse}}\}\{\hat{\tau}'\}(\hat{a}.\hat{e}')$ by assumption

(4) $\hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'\ \mathsf{type}$ by assumption

(5) , by assumption

(6) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{uetsm}(\tau'; e_{\text{parse}}); \hat{\Phi}} \hat{e}' \rightsquigarrow e \Leftarrow \tau$ by assumption

(7) $\Delta \vdash \Psi\ \mathsf{ueTSMs}$ by assumption

(8) $\Delta \vdash \tau'\ \mathsf{type}$ by Lemma 3.12 to (4)

(9) $\Delta \vdash \Psi, \hat{a} \hookrightarrow \mathtt{uetsm}(\tau';\ e_{\text{parse}})\ \mathsf{ueTSMs}$ by Definition 4.14 on (7), (8) and (5)

(10) $\Delta\ \Gamma \vdash e : \tau$ — by IH, part 1(b)(i) on (6)

Case (7.2j). We have:

(1) $\hat{e} = \texttt{uapuetsm}[b]\,[\hat{a}]$ — by assumption
(2) $\hat{\Psi} = \hat{\Psi}', \hat{a} \rightsquigarrow a \hookrightarrow \texttt{uetsm}(\tau;\, e_{\text{parse}})$ — by assumption
(3) $b \downarrow e_{\text{body}}$ — by assumption
(4) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}\ \texttt{Success} \cdot e_{\text{cand}}$ — by assumption
(5) $e_{\text{cand}} \uparrow_{\mathsf{CEExp}} \grave{e}$ — by assumption
(6) $\emptyset\ \emptyset \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e \Leftarrow \tau$ — by assumption
(7) $\emptyset \cap \Delta = \emptyset$ — by finite set intersection identity
(8) $\emptyset \cap \mathrm{dom}(\Gamma) = \emptyset$ — by finite set intersection identity
(9) $\emptyset \cup \Delta\ \emptyset \cup \Gamma \vdash e : \tau$ — 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 (7.2k). We have:

(1) $\hat{e} = \texttt{uelit}[b]$ — by assumption
(2) $\hat{\Psi} = \langle \mathcal{A}; \Psi, a \hookrightarrow \texttt{uetsm}(\tau;\, e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle$ — by assumption
(3) $b \downarrow e_{\text{body}}$ — by assumption
(4) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{inj}\ \texttt{Success} \cdot e_{\text{cand}}$ — by assumption
(5) $e_{\text{cand}} \uparrow_{\mathsf{CEExp}} \grave{e}$ — by assumption
(6) $\emptyset\ \emptyset \vdash^{\hat{\Delta};\hat{\Gamma};\langle \mathcal{A};\Psi,a\hookrightarrow\texttt{uetsm}(\tau;e_{\text{parse}});\mathcal{I}\uplus\tau\hookrightarrow a\rangle;\hat{\Phi};b} \grave{e} \rightsquigarrow e \Leftarrow \tau$ — by assumption
(7) $\emptyset \cap \Delta = \emptyset$ — by finite set intersection identity
(8) $\emptyset \cap \mathrm{dom}(\Gamma) = \emptyset$ — by finite set intersection identity
(9) $\emptyset \cup \Delta\ \emptyset \cup \Gamma \vdash e : \tau$ — 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 (7.2l). We have:

(1) $\hat{e} = \texttt{usyntaxup}\{e_{\text{parse}}\}\{\hat{\tau}'\}(\hat{a}.\hat{e}')$ — by assumption
(2) $\hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'\ \mathsf{type}$ — by assumption
(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi},\hat{a}\rightsquigarrow a\hookrightarrow\texttt{uptsm}(\tau';e_{\text{parse}})} \hat{e}' \rightsquigarrow e \Leftarrow \tau$ — by assumption
(4) $\Delta\ \Gamma \vdash e : \tau$ — by IH, part 1(b)(i) on (3)

Case (7.2m)**.** We have:

(1) $\hat{e} = \texttt{uimplicitp[}\hat{a}\texttt{](}\hat{e}\texttt{)}$ by assumption
(2) $\hat{\Phi} = \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{uptsm(}\tau'\texttt{;}\ e_{\text{parse}}\texttt{)}; \mathcal{I} \rangle$ by assumption
(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{uptsm(}\tau'\texttt{;}\ e_{\text{parse}}\texttt{)}; \mathcal{I} \uplus \tau \hookrightarrow a \rangle} \hat{e} \rightsquigarrow e \Leftarrow \tau$ 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 (7.4)**.** We have:

(1) $\hat{r} = \texttt{urule(}\hat{p}\texttt{.}\hat{e}\texttt{)}$ by assumption
(2) $r = \texttt{rule(}p\texttt{.}e\texttt{)}$ by assumption
(3) $\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\mid \langle \mathcal{A}'; \Upsilon \rangle$ by assumption
(4) $\hat{\Delta}\ \langle \mathcal{A} \uplus \mathcal{A}'; \Gamma \cup \Upsilon \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'$ by assumption
(5) $\Delta \vdash p : \tau \dashv\mid \Upsilon$ by Theorem 7.6, part 1 on (3)
(6) $\Delta\ \Gamma \cup \Upsilon \vdash e : \tau'$ by IH, part 1(b)(i) on (4)
(7) $\Delta\ \Gamma \vdash \texttt{rule(}p\texttt{.}e\texttt{)} : \tau \Mapsto \tau'$ by Rule (4.2) 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 (7.6a)**.** We have:

(1) $e = x$ by assumption
(2) $\Gamma = \Gamma', x : \tau$ by assumption
(3) $\Delta\ \Gamma', x : \tau \vdash x : \tau$ by Rule (4.1a)

Case (7.6b)**.** We have:

(1) $\grave{e} = \texttt{ceasc\{}\grave{\tau}\texttt{\}(}\grave{e}'\texttt{)}$ 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) $\Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau} \rightsquigarrow \tau\ \mathsf{type}$ by assumption
(5) $\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e}' \rightsquigarrow e \Leftarrow \tau$ by assumption
(6) $\Delta \cup \Delta_{\text{app}} \vdash \tau\ \mathsf{type}$ by Lemma 3.19 on (4)
(7) $\Delta\ \Gamma \vdash e : \tau$ by IH, part 2(b)(i) to (5), (2), (3) and (6)

Case (7.6c) through (7.6k)**.** In each of these cases, we apply:

- Lemma 3.19 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 (4.1), Rule (4.2), Lemma 4.1, the identification convention and exchange as needed.

Case (7.6l)**.** We have:

(1)	$\grave{e} = \texttt{cesplicede}[m;n]$	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 4.1 over Δ and Γ and exchange on (6)

ii. We induct on the assumption. There is one case.

Case (7.8)**.** We have:

(1)	$\grave{r} = \texttt{cerule}(p.\grave{e})$	by assumption
(2)	$r = \texttt{rule}(p.e)$	by assumption
(3)	$\Delta \vdash p : \tau \dashv\Vert\ \Upsilon$	by assumption
(4)	$\Delta\ \Gamma \cup \Upsilon \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e \Rightarrow \tau'$	by assumption
(5)	$\Delta \cap \Delta_{\text{app}} = \emptyset$	by assumption
(6)	$\text{dom}(\Gamma) \cap \text{dom}(\Upsilon) = \emptyset$	by identification convention
(7)	$\text{dom}(\Gamma_{\text{app}}) \cap \text{dom}(\Upsilon) = \emptyset$	by identification convention
(8)	$\text{dom}(\Gamma) \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset$	by assumption
(9)	$\text{dom}(\Gamma \cup \Upsilon) \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset$	by standard finite set definitions and identities on (6), (7) and (8)
(10)	$\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Upsilon \cup \Gamma_{\text{app}} \vdash e : \tau'$	by IH, part 2(a)(i) on (4), (5) and (9)
(11)	$\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \cup \Upsilon \vdash e : \tau'$	by exchange of Υ 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 (4.2) on (3) and (11)

(b) i. We induct on the assumption.

Case (7.7a)**.** 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 (7.7b) through (7.2h)**.** In each of these cases, we apply:

- Lemma 3.19 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 (4.1), Rule (4.2), Lemma 4.1, the identification convention and exchange as needed.

Case (7.7i)**.** We have:

(3)	$\grave{e} = \texttt{cesplicede}[m;n]$	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 4.1 over Δ and Γ and exchange on (9)

ii. We induct on the assumption. There is one case.

Case (7.9)**.** We have:

(1)	$\grave{r} = \texttt{cerule}(p.\grave{e})$	by assumption
(2)	$r = \texttt{rule}(p.e)$	by assumption
(3)	$\Delta \vdash p : \tau \dashv\mid \mathcal{Y}$	by assumption
(4)	$\Delta\ \Gamma \cup \mathcal{Y} \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e \Leftarrow \tau'$	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)	$\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\mathcal{Y}) = \emptyset$	by identification convention
(9)	$\mathrm{dom}(\Gamma_{\mathrm{app}}) \cap \mathrm{dom}(\mathcal{Y}) = \emptyset$	by identification convention
(10)	$\mathrm{dom}(\Gamma \cup \mathcal{Y}) \cap \mathrm{dom}(\Gamma_{\mathrm{app}}) = \emptyset$	by standard finite set definitions and identities on (8), (9) and (6)
(11)	$\Delta \cup \Delta_{\mathrm{app}}\ \Gamma \cup \mathcal{Y} \cup \Gamma_{\mathrm{app}} \vdash e : \tau'$	by IH, part 2(b)(i) on (4), (7), (10) and (5)
(12)	$\Delta \cup \Delta_{\mathrm{app}}\ \Gamma \cup \Gamma_{\mathrm{app}} \cup \mathcal{Y} \vdash e : \tau'$	by exchange of $\mathcal{Y}$ and Γ_{app} on (11)

(13) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash \texttt{rule}(p.e) : \tau \Rightarrow\!\!\!\!\Rightarrow \tau'$ by Rule (4.2) 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 (7.1m) in the proof of part 1(a)(i) and Case (7.2k) 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 (7.6l) in the proof of part 2(a)(i) and Case (7.7i) 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 $\hat{e}$,

$$\begin{aligned}
\|\hat{x}\| &= 0 \\
\|\texttt{uasc}\{\hat{\tau}\}(\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{uletval}(\hat{e}; \hat{x}.\hat{e}')\| &= \|\hat{e}\| + \|\hat{e}'\| \\
\|\texttt{uanalam}(\hat{x}.\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{ulam}\{\hat{\tau}\}(\hat{x}.\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{uap}(\hat{e}_1; \hat{e}_2)\| &= \|\hat{e}_1\| + \|\hat{e}_2\| \\
\|\texttt{utlam}(\hat{t}.\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{utap}\{\hat{\tau}\}(\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{ufold}(\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{uunfold}(\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{utpl}[L](\{i \hookrightarrow \hat{e}_i\}_{i \in L})\| &= \sum_{i \in L} \|\hat{e}_i\| \\
\|\texttt{upr}[\ell](\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{uin}[\ell](\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{umatch}[n](\hat{e}; \{\hat{r}_i\}_{1 \leq i \leq n})\| &= \|\hat{e}\| + \sum_{1 \leq i \leq n} \|r_i\| \\
\|\texttt{usyntaxue}\{e_{\text{parse}}\}\{\hat{\tau}\}(\hat{a}.\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{uimplicite}[\hat{a}](\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{uapuetsm}[b][\hat{a}]\| &= \|b\|
\end{aligned}$$

$$\|\texttt{uelit}[b]\| = \|b\|$$
$$\|\texttt{usyntaxup}\{e_{\text{parse}}\}\{\hat{\tau}\}(\hat{a}.\hat{e})\| = \|\hat{e}\|$$
$$\|\texttt{uimplicitp}[\hat{a}](\hat{e})\| = \|\hat{e}\|$$

and $\|r\|$ is defined as follows:

$$\|\texttt{urule}(\hat{p}.\hat{e})\| = \|\hat{e}\|$$

Going from part 1 to part 2, the metric remains stable:

$$\begin{aligned}
&\|\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{uapuetsm}[b][\hat{a}] \rightsquigarrow e \Rightarrow \tau\| \\
=&\|\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{uelit}[b] \rightsquigarrow e \Leftarrow \tau\| \\
=&\|\emptyset\ \emptyset \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e \Leftarrow \tau\| \\
=&\|b\|
\end{aligned}$$

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:

$$\|\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau\| < \|\Delta\ \Gamma \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \texttt{cesplicede}[m;n] \rightsquigarrow e \Rightarrow \tau\|$$

and

$$\|\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau\| < \|\Delta\ \Gamma \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \texttt{cesplicede}[m;n] \rightsquigarrow e \Leftarrow \tau\|$$

i.e. by the definitions above,

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

This is established by appeal to Condition 3.22, 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 3.22 and 7.9, we have that $\|\hat{e}\| < \|b\|$ as needed.

- In Case (7.2a) 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 (7.7a) 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 [60].) 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 [59].

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 [53]) 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] Ocaml batteries included. http://batteries.forge.ocamlcore.org/. Retrieved May 31, 2016., . 1.1.2

[2] core - jane street capital's standard library overlay. https://github.com/janestreet/core/. Retrieved May 31, 2016., . 1.1.2

[3] [Rust] Macros. https://doc.rust-lang.org/book/macros.html. Retrieved Nov. 3, 2015. 3.1.2

[4] Single sorted abstract binding trees in SML. https://github.com/cyrus-/sml-abt-single-sorted. Retrieved May 31, 2016., . 1.1.2

[5] Multi-sorted nominal abstract binding trees. https://github.com/RedPRL/sml-typed-abts. Retrieved May 31, 2016., . 1.1.2

[6] SML/NJ Quote/Antiquote. http://www.smlnj.org/doc/quote.html, . 2.4.2

[7] The Visible Compiler. http://www.smlnj.org/doc/Compiler/pages/compiler.html, . 2.4.6

[8] 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

[9] OWASP Top 10 2013. https://www.owasp.org/index.php/Top_10_2013-Top_10, 2013. 3

[10] 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

[11] Peter Aczel. A general Church-Rosser theorem. Technical report, Technical Report, University of Manchester, 1978. 1.1

[12] 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

[13] Alan Bawden. Quasiquotation in Lisp. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 4–12, 1999. URL http://repository.readscheme.org/ftp/papers/pepm99/bawden.pdf. 2.4.6

[14] George EP Box and Norman R Draper. *Empirical model-building and response surfaces*. John Wiley & Sons, 1987. 2.2

[15] 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

[16] Eugene Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *Proceedings of the 4th Workshop on Scala (SCALA '13)*, pages 3:1–3:10, 2013. 3, 2.4.6

[17] 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

[18] 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

[19] 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

[20] 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, 2.4.5

[21] 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

[22] Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In Greg Morrisett and Tarmo Uustalu, editors, *ICFP*, pages 429–442. ACM, 2013. ISBN 978-1-4503-2326-0. URL http://dl.acm.org/citation.cfm?id=2500365. 1.3.1

[23] Sebastian Erdweg and Felix Rieger. A framework for extensible languages. In *GPCE '13*, pages 3–12, 2013. 1.2.1, 2.4.5

[24] 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

[25] 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

[26] 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

[27] 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

[28] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, Cambridge, Mass., 2009. 2.4.5

[29] 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

[30] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium 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, 2.4.5

[31] 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. 2.4.6

[32] 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

[33] 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

[34] 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

[35] 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

[36] Robert Harper. *Programming in Standard ML*. 1997. URL http://www.cs.cmu.edu/~rwh/smlbook/book.pdf. Working draft, retrieved June 21, 2015. 1.1.2, 2.1, 5.1.2

[37] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012. 4.2.1, 4.2.3

[38] Robert Harper. *Practical Foundations for Programming Languages*. Second edition, 2015. URL https://www.cs.cmu.edu/~rwh/plbook/2nded.pdf. (Working Draft, Retrieved Nov. 19, 2015). 1.1, 2.1, 3.1.2, 3.2.1, 3.4, 3.2.2, 4.2.2

[39] T. P. Hart. MACRO definitions for LISP. Report A. I. MEMO 57, Massachusetts

Institute of Technology, A.I. Lab., Cambridge, Massachusetts, October 1963. 2.4.6

[40] 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

[41] David Herman. *A Theory of Typed Hygienic Macros*. PhD thesis, Northeastern University, Boston, MA, May 2010. 2.4.6

[42] John E Hopcroft. *Introduction to automata theory, languages, and computation*. Pearson Education India, 1979. 2.4.5

[43] 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. 2.4.5

[44] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *Journal of functional programming*, 8(04):437–444, 1998. 2.4.5

[45] 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

[46] 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

[47] 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. 2.4.6

[48] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 4.01 Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, September 2013. 1.2.1, 2.4.2, 2.4.5, 2.4.5

[49] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *ACM Sigplan Notices*, volume 9, pages 50–59. ACM, 1974. 2.3.2

[50] 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

[51] 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

[52] 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, 5.1.2

[53] 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

[54] J. McCarthy. History of lisp. In *History of programming languages I*, pages 173–185. ACM, 1978. 3

[55] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997. 1.1.2

[56] 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

[57] 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. 2.4.5

[58] 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

[59] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely composable type-specific languages. In *ECOOP '14*, 2014. 3.1, 8.6

[60] Cyrus Omar, Chenglong Wang, and Jonathan Aldrich. Composable and hygienic typed syntax macros. In *ACM Symposium on Applied Computing (SAC '15)*, 2015. 3.1, 8.3

[61] J.F. Pane and B.A. Myers. *Usability issues in the design of novice programming systems*. Citeseer, 1996. 1.2.2, 2.2

[62] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. 2.1, 3.2.1

[63] J C Reynolds. Types, abstraction and parametric polymorphism. In *IFIP'83, Paris, France*. North-Holland, September 1983. 5

[64] 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

[65] 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, 2.4.5, 1

[66] Denys Shabalin, Eugene Burmako, and Martin Odersky. Quasiquotes for Scala. Technical Report EPFL-REPORT-185242, 2013. 1.1.2, 2.4.6

[67] 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

[68] 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

[69] D. R. Tarditi and A. W. Appel. *ML-Yacc„version 2.0 Documentation*, 1990. 2.4.5

[70] 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

[71] 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

[72] Martin P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15 (4):147–161, 1994. 1.2.1

[73] 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, 2.4.5