Modularly Programmable Syntax

Cyrus Omar

TODO: get TR number

January 18, 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: TODO: keywords

TODO: dedication

Abstract

Full-scale functional programming languages often make *ad hoc* choices in the design of their textual syntax. For example, nearly all major functional languages build in derived forms for lists, but introducing derived forms for other library constructs, e.g. for HTML elements or regular expressions, requires forming new syntactic dialects of these languages. Programmers cannot combine syntactic dialects in general, limiting the choices ultimately available to them. In this work, we introduce and formally specify language primitives that mitigate the need for *ad hoc* derived forms and syntactic dialects by giving library providers the ability to programmatically control the expansion of certain *generalized literal forms* while maintaining a type discipline, a binding discipline and compositional reasoning principles.

Acknowledgments

TODO: Acknowledgments

Contents

List of Figures

Chapter 1

Introduction

> *The recent development of programming languages suggests that the simultaneous achievement of simplicity and generality in language design is a serious unsolved problem.*
>
> — John Reynolds (1970) [35]

1.1 Motivation

Programming languages come in many sizes. Small languages – i.e. "formal calculi" – allow language designers to study the mathematical properties of language primitives of interest in isolation. These studies then inform the design of "full-scale" languages, which combine several such primitives, or generalizations thereof.

Because small-scale languages are of interest mainly as objects of mathematical study, their designers often choose to specify only the abstract syntax of their primitives (or, when typesetting documents, stylized representations thereof). Full-scale languages, on the other hand, are both interesting objects of mathematical study and, ideally, useful for write large programs, so they typically also specify a more "programmer-friendly" textual concrete syntax that features various *derived syntactic forms*, i.e. forms defined by a context-independent "desugaring" to the set of base forms, that decrease the syntactic cost of certain common idioms. For example, Standard ML (SML) [17, 28], OCaml [25] and Haskell [23] build in derived forms that decrease the syntactic cost of working with lists. In these languages, the form `[1, 2, 3, 4, 5]` desugars to:

```
Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Nil)))))
```

The hope amongst many language designers is that a limited number of derived forms like these will suffice to produce a "general-purpose" programming language, i.e. one that satisfies programmers working in a wide variety of application domains. Unfortunately, a stable language design that fully achieves this ideal has yet to emerge, as evidenced by the diverse array of *syntactic dialects* – dialects that introduce only new derived forms – that continue to proliferate around all major contemporary languages. For example, Ur/Web is a syntactic dialect of Ur (an ML-like full-scale language [8]) that builds in derived forms for SQL queries, HTML elements and other datatypes used in the domain of web programming [9]. We will consider a large number of other examples of syntactic dialects in Sec. 2.2. Tools like Camlp4 [25], Sugar* [11, 12] and Racket's preprocessor [13], which we will discuss in Sec. 2.3, have decreased the engineering costs of constructing syntactic dialects, further contributing to their proliferation.

1.1.1 Dialects Considered Harmful

Some view this proliferation of dialects as harmless or even as desirable, arguing that programmers can simply choose the right dialect for the job at hand [42]. However, this "dialect-oriented" approach is, in an important sense, anti-modular: programmers cannot always "combine" different dialects when they want to use the primitives that they feature together within a single program. For example, a programmer might have access to a dialect featuring HTML syntax and to a dialect featuring regular expression syntax, but it is not always straightforward to, from these, construct a dialect featuring both. Both HTML and regular expression syntax might be useful when constructing, for example, a web-based bioinformatics tool.

In some cases, constructing the desired "combined dialect" is difficult simply because the constituent dialects are specified using different formalisms. In other cases, the constituent dialects may be specified using a formalism that does not operationalize the notion of dialect combination (e.g. Racket's preprocessor [13]). But even if we restrict our interest to dialects specified using a formalism that does operationalize some notion of dialect combination (or, equivalently, one that allows programmers to combine "dialect fragments"), there may still be a problem: the formalism may not guarantee that the combined dialect will conserve important properties that can be established about the dialects in isolation. For example, consider two syntactic dialects specified using Camlp4, one specifying derived syntax for finite mappings, the other specifying overlapping syntax for *ordered* finite mappings. Though each dialect has a deterministic grammar, when these grammars are naïvely combined, syntactic ambiguities will arise. We are aware of only one formalism that guarantees that determinism is conserved when syntactic dialects are combined [36], but it has limited expressive power, as we will discuss in Sec. 2.3.2.

1.1.2 Large Languages Considered Harmful

Dialects do sometimes have a less direct influence on large-scale software development: they can help convince the designers in control of comparatively popular languages, like OCaml and Scala, to include some variant of the primitives that they feature into backwards-compatible language revisions. This *ad hoc* approach is unsustainable, for three main reasons. First, as we will demonstrate in Sec. 2.2, there are simply too many potentially useful such primitives, and many of these capture idioms common only in relatively narrow application domains. It is unreasonable to expect language designers to be able to evaluate all of these use cases in a timely and informed manner. Second, primitives introduced earlier in a language's lifespan can end up monopolizing finite "syntactic resources", forcing subsequent primitives to use ever more esoteric forms. And third, primitives that prove after some time to be flawed in some way cannot be removed or modified without breaking backwards compatibility. For these reasons, language designers are justifiably reticent to add new primitives to major languages.

1.1.3 Toward More General Primitives

This leaves two possible paths forward. One is to simply eschew "niche" primitives and settle on the existing designs, which might be considered to sit at a "sweet spot" in the overall language design space (accepting that in some circumstances, this leads to high syntactic cost). The other path forward is to search for a small number of highly general primitives that allow us degrade many of the constructs that are built primitively into languages and their dialects today instead to modular library constructs. Encouragingly, primitives of this sort do occasionally arise. For example, a recent revision of OCaml added support for generalized algebraic data types (GADTs), based on research on guarded recursive datatype constructors [43]. Using GADTs, OCaml was able to move some of the *ad hoc* machinery for typechecking operations that use format strings, like `sprintf`, out of the language and into a library. Syntactic machinery related to `sprintf`, however, remains built in.

1.2 Overview of Contributions

Our aim in this work is to introduce primitive language constructs that reduce the need for syntactic dialects and *ad hoc* derived syntactic forms. In particular, we introduce **typed syntax macros**, or **TSMs**. TSMs are applied like functions to *generalized literal forms* and programmatically control their parsing and expansion. This occurs statically (i.e. simultaneously with typing). We introduce TSMs first for a simple language of expressions and types in Chapter 3, then add support for pattern matching in Chapter 4 and type and module parameters in Chapter 6.

In Chapter 5 and Chapter 6, we also show how library clients can designate, for any type, a privileged TSM at that type, and then rely on local type inference to invoke that TSM and apply its parameters implicitly. TSM implicits can reduce the syntactic cost of an idiom to very nearly the same extent that a special-purpose dialect can, while avoiding the problems described above.

As vehicles for this work, we will specify a small-scale typed lambda calculus in each of the chapters just mentioned, each building upon the previous one. For the sake of examples, we will also describe (but not formally specify) a full-scale functional language called VerseML.[1] VerseML is, as its name suggests, a dialect of ML. It diverges from other dialects of ML that have a similar underlying type structure – in particular,

[1]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.

OCaml – in that it uses a local type inference scheme [34] (like, for example, Scala [29]) for reasons that have to do with the mechanisms described in Chapter 5. The reason we will not follow Standard ML [28] in giving a complete formal specification of VerseML in this work is both to emphasize that the primitives we introduce are fairly insensitive to the details of the underlying type structure of the language (so TSMs can be considered for inclusion in a variety of languages, not only dialects of ML), and to avoid distracting the reader (and the author) with specifications of primitives that are already well-understood in the literature and that are orthogonal to those that are the focus of this work.

The main challenge will come in maintaining the following:

- a *type discipline*, meaning that the language must be type safe, and that programmers examining a well-typed expression must be able to determine its type without examining its expansion;
- a *hygienic binding discipline*, meaning that the expansion logic must not be permitted to make "hidden assumptions" about the names of variables at macro application sites, nor introduce "hidden bindings" into other terms; and
- *compositional reasoning principles*, meaning that library providers must have the ability to reason about the syntax that they have defined in isolation, and clients must be able to use macros safely in any combination, without the possibility of conflict.[2]

We will, of course, make these notions more technically precise as we continue.

Thesis Statement

In summary, this work defends the following statement:

> A functional programming language can give library providers the ability to express new syntactic expansions while maintaining a type discipline, a hygienic binding discipline and compositional reasoning principles.

1.3 Disclaimers

Before we continue, it may be prudent to explicitly acknowledge that completely eliminating the need for dialects would indeed be asking for too much: certain language design decisions are fundamentally incompatible with others or require coordination across a language design. We aim only to decrease the need for syntactic dialects in this work. We will not consider situations that require modifications to the underlying type structure of a language (though this is a rich avenue for future work).

It may also be useful to explicitly acknowledge that library providers could leverage the primitives we introduce to define constructs that are in "poor taste". We expect that

[2] This is not quite true – name clashes of the usual sort can arise. We will tacitly assume that in practice, they can be avoided extrinsically, e.g. by using a URI-based naming scheme as in the Java ecosystem.

in practice, VerseML will come with a standard library defining an expertly curated collection of standard constructs, as well as guidelines for advanced users regarding when it would be sensible to use the mechanisms we introduce (following the example of languages that support operator overloading or type classes [16], which also have some potential for "abuse" or "overuse").

Chapter 2

Background

2.1 Preliminaries

This work is rooted in the tradition of full-scale functional languages with non-trivial type structure like ML and Haskell (as might have been obvious from the exposition in Chapter 1). Familiarity with basic concepts in these languages, e.g. variables, types, functions, tuples, records, recursive datatypes and nested 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* [17] (a working draft can be found online). We briefly discuss integration of TSMs and TSLs into languages from other language design traditions in Sec. 8.6.3.

Chapter 6 and Chapter 5, and some of the motivating examples given below, consider questions of integration with an ML-style module system, so readers with experience in a language without such a module system (e.g. Haskell) are also advised to review the relevant chapters in *Programming in Standard ML* [17] before delving into these portions of these chapters.

The formal systems that we will construct in later chapters are specified within the metatheoretic framework of type theory. More specifically, we assume familiarity with fundamental background concepts (e.g. abstract binding trees, substitution, implicit identification of terms up to α-equivalence, structural induction and rule induction) covered in detail in Harper's *Practical Foundations for Programming Languages, Second Edition* (*PFPL*) [19] (a working draft can be found online). Familiarity with other formal accounts of type systems, e.g. Pierce's *Types and Programming Languages* (*TAPL*) [33], should also suffice. This document is organized so as to be readable even if the sections describing formal systems are skipped (although some precision will, of course, be lost).

2.2 Motivating Examples

In Chapter 1, we gave the example of derived list syntax in languages like SML, OCaml and Haskell. To further motivate our contributions, we now provide more examples where defining new derived syntactic forms could decrease the syntactic cost of working with certain data structures. We cover the first example – regular expressions, expressed both using recursive sum types and abstract types – in substantial detail. We will refer back to this example in subsequent chapters. We then more concisely survey a number of other examples, grouped into categories, to establish the broad applicability of our contributions.

2.2.1 Regular Expressions

Regular expressions, or *regexes*, specify string patterns (of a certain class) [40]. They are particularly common in domains like natural language processing and bioinformatics. The abstract syntax of regexes, r, over strings, s, is specified below:

$$r ::= \mathbf{empty} \mid \mathbf{str}(s) \mid \mathbf{seq}(r;r) \mid \mathbf{or}(r;r) \mid \mathbf{star}(r)$$

Recursive Sums One way to express this abstract syntax is by defining a recursive sum type [19]. In VerseML, a recursive labeled sum type can be defined like this:

```
type Rx = Empty | Str of string | Seq of Rx * Rx |
          Or of Rx * Rx | Star of Rx
```

Figure 2.1: Definition of the recursive labeled sum type `Rx`

Values of type `Rx` are constructed by applying a label to a value of the type specified in the corresponding case above (or simply using the label by itself if no corresponding type is specified). For example, we can define a regular expression that matches only the strings "A", "T", "G" and "C" as follows:

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

This is too verbose to be practical in all but the simplest examples, so the POSIX standard specifies a more concise concrete syntax for regexes [3]. Several programming languages support derived syntax for regexes based on this standard, e.g. Perl [10]. Let us consider a hypothetical dialect of ML called ML+Rx (perhaps constructed using a tool like Camlp4, discussed in Sec. 2.3.2) that similarly builds in derived forms for regexes. We will compare VerseML to ML+Rx in later chapters. ML+Rx extends the concrete syntax of ML with support for *regex expression literals* delimited by forward slashes. For example, the following regex literal desugars to the expression above:

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

ML+Rx also supports *spliced subexpressions* in regex literals, so that regexes can be constructed from other values. For example, the function `example_rx` shown in Figure 2.2 constructs a regex by splicing in a string, `name`, and another regex, `ssn`. The prefix `@` followed by the variable `name` indicates that the expression `name` should be spliced in as a string, and the prefix `%` followed by the variable `ssn` indicates that `ssn` should be spliced in as a regex. The body of `example_rx` desugars to the following:

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

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

Figure 2.2: Spliced subexpressions in ML+Rx

Notice that `name` appears wrapped in the constructor `Str` because it was prefixed by `@`, whereas `ssn` appears unadorned because it was prefixed by `%`.

To splice in an expression that does not take the form of a variable, e.g. a function call, we can delimit it with parentheses:

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

Finally, ML+Rx allows us to pattern match over a value of type `Rx` using analagous derived pattern syntax. For example, the body of the following function reads the name and social security number back out of a regex generated by the function `example_rx`:

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

Figure 2.3: Derived pattern syntax in ML+Rx

This expression desugars to:

```
fun read_example_rx(r : Rx) : (string * Rx) option =>
  match r with
    Seq(Str(name), Seq(Str ": ", ssn)) => Some (name, ssn)
  | _ => None
```

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, $\mathbf{seq}(\mathbf{empty}, r)$ has normal form r. It can be useful for regexes with the same normal form to be indistinguishable from the perspective of client code. Second, it can be useful for performance reasons to maintain additional data alongside regexes (e.g. a corresponding finite automaton), but one would not want to expose this "implementation detail" to clients. In fact, there may be many ways to represent regular expression patterns, each with different performance trade-offs, so we would like to provide clients with a choice of implementations. For these reasons, another approach in VerseML, as in ML, is to abstract over the choice of representation using the module system's support for abstract types. In particular, we can define the *module signature* `RX`, shown in Figure 2.4, where the type of patterns, `t`, is held abstract.

Clients of any module `R` that has been sealed against `RX`, written `R :> RX`, manipulate patterns as values of the type `R.t` using the interface described by this signature. The identity of the type `R.t` is held abstract outside the module during typechecking (i.e.

```
signature RX = sig {
  type t
  val Empty : t
  val Str : string -> t
  val Seq : t * t -> t
  val Or : t * t -> t
  val Star : t -> t
  val Group : t -> t
  val case : (
    t -> {
      Empty : 'a,
      Str : string -> 'a,
      Seq : t * t -> 'a,
      Or : t * t -> 'a,
      Star : t -> 'a,
      Group : t -> 'a
    } -> 'a
}
```

Figure 2.4: Definition of the `RX` signature

it acts as a newly generated type). As a result, the burden of proving that there is no way to use the case analysis function to distinguish patterns with the same normal form is local to the module, and implementation details do not escape (and can thus evolve freely).

TODO: talk about module-parameterized derived syntactic forms for this

TODO: talk about pattern matching over values of abstract type

2.2.2 Lists, Sets, Maps, Vectors and Other Containers

TODO: write this (Spring 2016)

2.2.3 HTML and Other Web Languages

2.2.4 Dates, URLs and Other Standardized Formats

2.2.5 Query Languages

The language of regular expressions can be considered a query language over strings. There are many other query languages that focus on different types of data, e.g. XQuery for XML trees, or that are associated with various database technologies, e.g. SQL for relational databases. TODO: finish this (Spring 2016)

2.2.6 Monadic Commands

TODO: write this; cite Bob's blog (Spring 2016)

TODO: http://www.cs.umd.edu/ mwh/papers/monadic.pdf

2.2.7 Quasiquotation and Object Language Syntax

TODO: write this (Spring 2016)

2.2.8 Grammars

TODO: write this (Spring 2016)

2.2.9 Mathematical and Scientific Notations

SMILES: Chemical Notation

TODO: write this; cite SMILES `https://en.wikipedia.org/wiki/Simplified_molecular-input_line-entry_system` (Spring 2016)

TeX Mathematical Formula Notation

TODO: write this (Spring 2016)

2.2.10 Others

Get examples from: `http://voelter.de/data/pub/mbeddr-cs-oopsla2015-preprint.pdf`

2.3 Existing Approaches

TODO: revise, reformat and extend (Spring 2016 / as needed)

2.3.1 Dynamic String Parsing

To expose this more concise concrete syntax for regular expression patterns to clients, the most common approach is to provide a function that parses strings to produce patterns. Because, as just mentioned, there may be many implementations of the `RX` signature, the usual approach is to define a parameterized module (a.k.a. a *functor* in SML) defining utility functions like this abstractly:

```
module RXUtil(R : RX) => mod {
  fun parse(s : string) : R.t => (* ... regex parser here ... *)
}
```

This allows a client of any module `R : RX` to use the following definitions:

```
let module RUtil = RXUtil(R)
let val rxparse = RUtil.parse
```

to construct patterns like this:

```
rxparse "A|T|G|C"
```

Unfortunately, this approach is imperfect for several reasons:

1. First, there are syntactic conflicts between string escape sequences and pattern escape sequences. For example, the following is not a well-formed term:

   ```
   let val ssn = rxparse "\d\d\d-\d\d-\d\d\d\d"
   ```

   When compiling an expression like this, the client would see an error message like `error: illegal escape character`[1], because `\d` is not a valid string escape sequence. In a small lab study, we observed that this class of error often confused even experienced programmers if they had not used regular expressions recently [30]. One workaround has higher syntactic cost – we must double all backslashes:

   ```
   let val ssn = rxparse "\\d\\d\\d-\\d\\d-\\d\\d\\d\\d"
   ```

   Some languages, anticipating such modes of use, build in alternative string forms that leave escape sequences uninterpreted. For example, OCaml supports the following, which has only a constant syntactic cost:

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

2. The next problem is that dynamic string parsing mainly decreases the syntactic cost of complete patterns. Patterns constructed compositionally cannot easily benefit from this technique. For example, consider the following function from strings to patterns:

   ```
   fun example(name : string) =>
     R.Seq(R.Str(name), R.Seq(rxparse ": ", ssn)) (* ssn as above *)
   ```

   Had we built derived syntax for regular expression patterns into the language primitively (following Unix conventions of using forward slashes as delimiters), we could have used *splicing syntax*:

   ```
   fun example_shorter(name : string) => /@name: %ssn/
   ```

   An identifier (or parenthesized expression, not shown) prefixed with an `@` is a spliced string, and one prefixed with a `%` is a spliced pattern.

   It is difficult to capture idioms like this using dynamic string parsing, because strings cannot contain sub-expressions directly.

3. For functions like `example` where we are constructing patterns on the basis of data of type `string`, using strings coincidentally to introduce patterns tempts programmers to use string concatenation in subtly incorrect ways. For example, consider the following seemingly more readable definition of `example`:

[1]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 example_bad(name : string) =>
  rxparse (name ^ {rx|: \d\d\d-\d\d-\d\d\d\d|rx})
```

Both `example` and `example_bad` have the same type and behave identically at many inputs, particularly "typical" inputs (i.e. alphabetic names). It is only when the input `name` contains special characters that have meaning in the concrete syntax of patterns that a problem arises.

In applications that query sensitive data, mistakes like this lead to *injection attacks*, which are among the most common and catastrophic security threats on the web today [4]. These are, of course, a consequence of the programmer making a mistake in an effort to decrease syntactic cost, but proving that mistakes like this have not been made involves reasoning about complex run-time data flows, so it is once again notoriously difficult to automate. If our language supported derived syntax for patterns, this kind of mistake would be substantially less common (because `example_shorter` has lower syntactic cost than `example_bad`).

4. The next problem is that pattern parsing does not occur until the pattern is evaluated. For example, the following malformed pattern will only trigger an exception when this expression is evaluated during the full moon:

```
case(moon_phase) {
    Full => rxparse "(GC" (* malformedness not statically detected *)
  | _ => (* ... *)
}
```

Though malformed patterns can sometimes be discovered dynamically via testing, empirical data gathered from large open source projects suggests that there remain many malformed regular expression patterns that are not detected by a project's test suite "in the wild" [38].

Statically verifying that pattern formation errors will not dynamically arise requires reasoning about arbitrary dynamic behavior. This is an undecidable verification problem in general and can be difficult to even partially automate. In this example, the verification procedure would first need to be able to establish that the variable `rxparse` is equal to the parse function `RUtil.parse`. If the string argument had not been written literally but rather computed, e.g. as `"(G" ^ "C"` where `^` is the string concatenation function applied in infix style, it would also need to be able to establish that this expression is equivalent to the string `"(GC"`. For patterns that are dynamically constructed based on input to a function, evaluating the expression statically (or, more generally, in some earlier "stage" of evaluation [22]) also does not suffice.

Of course, asking the client to provide a proof of well-formedness would defeat the purpose of lowering syntactic cost.

In contrast, were our language to primitively support derived pattern syntax, pattern parsing would occur at compile-time and so malformed patterns would produce a compile-time error.

5. Dynamic string parsing also necessarily incurs dynamic cost. Regular expression

patterns are common when processing large datasets, so it is easy to inadvertently incur this cost repeatedly. For example, consider mapping over a list of strings:

```
map exmpl_list (fn s => rxmatch (rxparse "A|T|G|C") s)
```

To avoid incurring the parsing cost for each element of `exmpl_list`, the programmer or compiler must move the parsing step out of the closure (for example, by eta-reduction in this simple example).[2] If the programmer must do this, it can (in more complex examples) increase syntactic cost and cognitive cost by moving the pattern itself far away from its use site. Alternatively, an appropriately tuned memoization (i.e. caching) strategy could be used to amortize some of this cost, but it is difficult to reason compositionally about performance using such a strategy.

In contrast, were our language to primitively support derived pattern syntax, the expansion would be computed at compile-time and incur no dynamic cost.

The problems above are not unique to regular expression patterns. Whenever a library encourages the use of dynamic string parsing to address the issue of syntactic cost (which is, fundamentally, not a dynamic issue), these problems arise. This fact has motivated much research on reducing the need for dynamic string parsing [6]. Existing alternatives can be broadly classified as being based on either *direct syntax extension* or *static term rewriting*. We describe these next, in Secs. 2.3.2 and 2.3.3 respectively.

2.3.2 Direct Syntax Extension

One tempting alternative to dynamic string parsing is to use a system that gives the users of a language the power to directly extend its concrete syntax with new derived forms.

The simplest such systems are those where the elaboration of each new syntactic form is defined by a single rewrite rule. For example, Gallina, the "external language" of the Coq proof assistant, supports such extensions [27]. A formal account of such a system has been developed by Griffin [15]. Unfortunately, a single equation is not enough to allow us to express pattern syntax following the usual conventions. For example, a system like Coq's cannot handle escape characters, because there is no way to programmatically examine a form when generating its expansion.

Other syntax extension systems are more flexible. For example, many are based on context-free grammars, e.g. Sugar* [11] and Camlp4 [25] (amongst many others). Other systems give library providers direct programmatic access to the parse stream, like Common Lisp's *reader macros* [39] (which are distinct from its term-rewriting macros, described in Sec. 2.3.3 below) and Racket's preprocessor [13]. All of these would allow us to add pattern syntax into our language's grammar, perhaps following Unix conventions and supporting splicing syntax as described above:

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

[2] Anecdotally, in major contemporary compilers, this optimization is not automatic.

We sidestep the problems of dynamic string parsing described above when we directly extend the syntax of our language using any of these systems. Unfortunately, direct syntax extension introduces serious new problems. First, the systems mentioned thus far cannot guarantee that syntactic conflicts between such extensions will not arise. As stated directly in the Coq manual: "mixing different symbolic notations in [the] same text may cause serious parsing ambiguity". If another library provider used similar syntax for a different implementation or variant of regular expressions, or for some other unrelated construct, then a client could not simultaneously use both libraries at the same time. So properly considered, every combination of extensions introduced using these mechanisms creates a *de facto* syntactic dialect of our language. The benefit of these systems is only that they lower the implementation cost of constructing syntactic dialects.

In response to this problem, Schwerdfeger and Van Wyk developed a modular analysis that accepts only context-free grammar extensions that begin with an identifying starting token and obey certain constraints on the follow sets of base language's nonterminals [36]. Extensions that specify distinct starting tokens and that satisfy these constraints can be used together in any combination without the possibility of syntactic conflict. However, the most natural starting tokens like `rx` cannot be guaranteed to be unique. To address this problem, programmers must agree on a convention for defining "globally unique identifiers", e.g. the common URI convention used on the web and by the Java packaging system. However, this forces us to use a more verbose token like `edu_cmu_VerseML_rx`. There is no simple way for clients of our extension to define scoped abbreviations for starting tokens because this mechanism operates purely at the level of the context-free grammar.

Putting this aside, we must also consider another modularity-related question: which particular module should the expansion use? Clearly, simply assuming that some module identified as `R` matching `RX` is in scope is a brittle solution. In fact, we should expect that the system actively prevents such capture of specific variable names to ensure that variables (here, module variables) can be freely renamed. Such a *hygiene discipline* is well-understood only when performing term-to-term rewriting (discussed below) or in simple language-integrated rewrite systems like those found in Coq. For mechanisms that operate strictly at the level of context-free grammars or the parse stream, it is not clear how one could address this issue. The onus is then on the library provider to make no assumptions about variable names and instead require that the client explicitly identify the module they intend to use as an "argument" within the newly introduced form:

```
let val ssn = edu_cmu_VerseML_rx R /\d\d\d-\d\d-\d\d\d\d/
```

Another problem with the approach of direct syntax extension is that, given an unfamiliar piece of syntax, there is no straightforward method for determining what type it will have, causing difficulties for both humans (related to code comprehension) and tools.

TODO: Related work I haven't mentioned yet:

- Fan: http://zhanghongbo.me/fan/start.html
- Well-Typed Islands Parse Faster:
  `http://www.ccs.neu.edu/home/ejs/papers/tfp12-island.pdf`

- User-defined infix operators
- SML quote/unquote
- That Modularity paper
- Template Haskell and similar

2.3.3 Term Rewriting

An alternative approach is to leave the concrete syntax of the language fixed, but repurpose it for novel ends using a *local term-rewriting system*. The LISP macro system [20] is perhaps the most prominent example of such a system. Early variants of this system suffered from the problem of unhygienic variable capture just described, but later variants, notably in the Scheme dialect of LISP, brought support for enforcing hygiene [24]. 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 [14, 21] and integrated into languages, e.g. MacroML [14] and Scala [7].

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 [2]). 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 [5] and Scala [37], 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 syntactic 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.

2.3.4 Active Libraries

The design we are proposing also has conceptual roots in earlier work on *active libraries*, which similarly envisioned using compile-time computation to give library providers more control over various aspects of a programming system, including its concrete (but did not take an approach rooted in the study of type systems) [41]. TODO: flesh this out and connect it to previous stuff

Chapter 3

Unparameterized Expression TSMs (ueTSMs)

We now introduce a new primitive – the **typed syntax macro** (TSM). TSMs, like term-rewriting macros (Sec. 2.3.3), 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 6.

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 2.1. We then formally specify ueTSMs with a reduced calculus, miniVerse$_{\mathbf{UE}}$, in Sec. 3.2.

3.1.1 Usage

In the following concrete VerseML expression, we apply a TSM named `$rx` to the *generalized literal form* `/A|T|G|C/`:

```
$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 [31] and multipart forms [32], 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 [1].

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. 7.2).

```
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 6, 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. 7.1. 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 [19], 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. 2.2.1, the parse function must be able to parse subexpressions out of the supplied literal body. For example, consider the code snippet in Figure 2.2, 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 examine 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. 2.3.3.)

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 6.

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. 2.2.1.

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 5 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 $\mathsf{miniVerse}_{\mathbf{UE}}$

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

3.2.1 Syntax of the Inner Core

The *inner core of* $\mathsf{miniVerse}_{\mathbf{UE}}$ 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* [19] (or another text on type systems, e.g. *TAPL* [33]) 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
			$\mathtt{parr}(\tau;\tau)$	$\tau \rightharpoonup \tau$	partial function
			$\mathtt{all}(t.\tau)$	$\forall t.\tau$	polymorphic
			$\mathtt{rec}(t.\tau)$	$\mu t.\tau$	recursive
			$\mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})$	$\langle\{i \hookrightarrow \tau_i\}_{i\in L}\rangle$	labeled product
			$\mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i\in L})$	$[\{i \hookrightarrow \tau_i\}_{i\in L}]$	labeled sum
Exp	e	$::=$	x	x	variable
			$\mathtt{lam}\{\tau\}(x.e)$	$\lambda x{:}\tau.e$	abstraction
			$\mathtt{ap}(e;e)$	$e(e)$	application
			$\mathtt{tlam}(t.e)$	$\Lambda t.e$	type abstraction
			$\mathtt{tap}\{\tau\}(e)$	$e[\tau]$	type application
			$\mathtt{fold}\{t.\tau\}(e)$	$\mathtt{fold}(e)$	fold
			$\mathtt{unfold}(e)$	$\mathtt{unfold}(e)$	unfold
			$\mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L})$	$\langle\{i \hookrightarrow e_i\}_{i\in L}\rangle$	labeled tuple
			$\mathtt{pr}[\ell](e)$	$e \cdot \ell$	projection
			$\mathtt{in}[L][\ell]\{\{i \hookrightarrow \tau_i\}_{i\in L}\}(e)$	$\ell \cdot e$	injection
			$\mathtt{case}[L]\{\tau\}(e;\{i \hookrightarrow x_i.e_i\}_{i\in L})$	$\mathtt{case}\ e\ \{i \hookrightarrow x_i.e_i\}_{i\in L}$	case analysis

Figure 3.4: Abstract syntax of types and expanded expressions, which form the *inner core of* 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`)` [19]. 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. $\mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})$, are identified up to mutual reordering of the label set and the corresponding argument sequence. 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:

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

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

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

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

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

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

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

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

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

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

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

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* [19]. 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* $\Delta \vdash [\tau'/t]\tau\ \mathsf{type}$ *and* $\Delta \vdash \tau'\ \mathsf{type}$ *then* $\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}$.
2. *If* $\Delta\ [\tau'/t]\Gamma \vdash [\tau'/t]e : [\tau'/t]\tau$ *and* $\Delta \vdash \tau'\ \mathsf{type}$ *then* $\Delta, t\ \mathsf{type}\ \Gamma \vdash e : \tau$.
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 $\Delta \vdash [\tau'/t]\tau\ \mathsf{type}$ 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* $\Delta \vdash \Gamma\ \mathsf{ctx}$ *then* $\Delta \vdash \tau\ \mathsf{type}$.

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* 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'
$e\ \mathsf{val}$	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* $e'\ \mathsf{val}$.

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* $e\ \mathsf{val}$ *then:*

1. *If* $\tau = \mathtt{parr(}\tau_1\mathtt{;}\tau_2\mathtt{)}$ *then* $e = \mathtt{lam\{}\tau_1\mathtt{\}(}x.e'\mathtt{)}$ *and* $x : \tau_1 \vdash e' : \tau_2$.

2. *If* $\tau = \texttt{all}(t.\tau')$ *then* $e = \texttt{tlam}(t.e')$ *and* $t\ \mathsf{type} \vdash e' : \tau'$.
3. *If* $\tau = \texttt{rec}(t.\tau')$ *then* $e = \texttt{fold}\{t.\tau'\}(e')$ *and* $\vdash e' : [\texttt{rec}(t.\tau')/t]\tau'$ *and* $e'\ \mathsf{val}$.
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* $e_i\ \mathsf{val}$ *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* $e'\ \mathsf{val}$.

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

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
			$\texttt{uin}[L][\ell]\{\{i \hookrightarrow \hat{\tau}_i\}_{i\in L}\}(\hat{e})$	$\ell\cdot\hat{e}$	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:

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

- 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:

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

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

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

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

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

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

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

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

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 $\hat{t} \in \text{dom}(\mathcal{D})$ to the hypothesis $\hat{t} \leadsto t$, for some type variable t. We write $\mathcal{D} \uplus \hat{t} \leadsto t$ for the type sigil expansion context that maps $\hat{t}$ to $\hat{t} \leadsto t$ and defers to $\mathcal{D}$ for all other type sigils (i.e. the previous mapping, if it exists, is updated). We define $\hat{\Delta}, \hat{t} \leadsto t \; \mathsf{type}$ when $\hat{\Delta} = \langle \mathcal{D}; \Delta \rangle$ as an abbreviation of

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

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, $\mathtt{uall}(\hat{t}.\mathtt{uall}(\hat{t}.\hat{t}))$:

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

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* $\langle \mathcal{D}; \Delta \rangle \vdash \hat{\tau} \rightsquigarrow \tau \; \mathsf{type}$ *then* $\Delta \vdash \tau \; \mathsf{type}$.

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 metafunction $\mathcal{U}(\Delta)$ which maps Δ onto a an unexpanded type formation context as follows:

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

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

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 $\hat{x} \in \mathrm{dom}(\mathcal{G})$ to the hypothesis $\hat{x} \rightsquigarrow x$, for some expression variable, x. We write $\mathcal{G} \uplus \hat{x} \rightsquigarrow x$ for the expression 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:

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

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

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

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

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

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

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

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:

$$\mathcal{U}(\emptyset) = \langle \emptyset; \emptyset \rangle$$
$$\mathcal{U}(\Gamma, x : \tau) = \mathcal{U}(\Gamma), \widehat{x} \leadsto x : \tau$$

  - 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* $\mathcal{U}(\Delta)\ \mathcal{U}(\Gamma) \vdash_{\hat{\Psi}} \mathcal{U}(e) \leadsto e : \tau$.

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. 7.2, 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):

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

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 $\hat{a} \in \text{dom}(\mathcal{A})$ 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 $a \in \text{dom}(\Psi)$ to an *expanded ueTSM definition*, $a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}})$, 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* $\emptyset\ \emptyset \vdash e_{parse} : \texttt{Body} \rightharpoonup \texttt{ParseResultExp}$.

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

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

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 $\texttt{uapuetsm}[b][\hat{a}]$.

The typed expansion rule governing ueTSM application is below:

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

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 $\texttt{Success} \cdot e_{\text{cand}}$ 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.

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

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
		$\texttt{cesplicedt}[m;n]$	$\texttt{spliced}\langle m, n\rangle$	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
		$\texttt{cein}[L][\ell]\{\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}\}(\grave{e})$	$\ell \cdot \grave{e}$	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
		$\texttt{cesplicede}[m;n]$	$\texttt{spliced}\langle m, n\rangle$	spliced

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

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:

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

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**
$\Delta \vdash^{\mathbb{T}} \grave{\tau} \leadsto \tau\ \mathsf{type}$	Candidate expansion type $\grave{\tau}$ is well-formed and has expansion τ assuming Δ and type splicing scene $\mathbb{T}$.
$\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e : \tau$	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*, $\Delta \vdash^{\mathbb{T}} \grave{\tau} \rightsquigarrow \tau\ \mathsf{type}$, 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.

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

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

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

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

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

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

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 $\texttt{cesplicedt}[m;n]$, 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:

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

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, $\langle \mathcal{D}; \Delta_{\text{app}} \rangle$, 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) $\Delta = \Delta', t \text{ type}$ — by assumption
(2) $\check{\tau} = t$ — by assumption
(3) $\tau = t$ — by assumption
(4) $\Delta', t \text{ type} \vdash t \text{ type}$ — by Rule (3.1a)
(5) $\Delta', t \text{ type} \cup \Delta_{\text{app}} \vdash t \text{ type}$ — by Lemma 3.2 over Δ_{app} to (4)

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

(1) $\check{\tau} = \texttt{ceparr}(\check{\tau}_1; \check{\tau}_2)$ — by assumption
(2) $\tau = \texttt{parr}(\tau_1; \tau_2)$ — by assumption
(3) $\Delta \vdash^{\langle \mathcal{D}; \Delta_{\text{app}} \rangle; b} \check{\tau}_1 \rightsquigarrow \tau_1 \text{ type}$ — by assumption
(4) $\Delta \vdash^{\langle \mathcal{D}; \Delta_{\text{app}} \rangle; b} \check{\tau}_2 \rightsquigarrow \tau_2 \text{ type}$ — by assumption
(5) $\Delta \cup \Delta_{\text{app}} \vdash \tau_1 \text{ type}$ — by IH on (3)
(6) $\Delta \cup \Delta_{\text{app}} \vdash \tau_2 \text{ type}$ — by IH on (4)
(7) $\Delta \cup \Delta_{\text{app}} \vdash \texttt{parr}(\tau_1; \tau_2) \text{ type}$ — by Rule (3.1b) on (5) and (6)

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

(1) $\check{\tau} = \texttt{ceall}(t.\check{\tau}')$ — by assumption
(2) $\tau = \texttt{all}(t.\tau')$ — by assumption
(3) $\Delta, t \text{ type} \vdash^{\langle \mathcal{D}; \Delta_{\text{app}} \rangle; b} \check{\tau}' \rightsquigarrow \tau' \text{ type}$ — by assumption
(4) $\Delta, t \text{ type} \cup \Delta_{\text{app}} \vdash \tau' \text{ type}$ — by IH on (3)
(5) $\Delta \cup \Delta_{\text{app}}, t \text{ type} \vdash \tau' \text{ type}$ — by exchange over Δ_{app} on (4)
(6) $\Delta \cup \Delta_{\text{app}} \vdash \texttt{all}(t.\tau') \text{ type}$ — 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) $\check{\tau} = \texttt{cesplicedt}[m; n]$ — by assumption
(2) $\mathsf{parseUTyp}(\mathsf{subseq}(b; m; n)) = \hat{\tau}$ — by assumption
(3) $\langle \mathcal{D}; \Delta_{\text{app}} \rangle \vdash \hat{\tau} \rightsquigarrow \tau \text{ type}$ — by assumption
(4) $\Delta_{\text{app}} \vdash \tau \text{ type}$ — by Lemma 3.12 on (3)
(5) $\Delta \cup \Delta_{\text{app}} \vdash \tau \text{ type}$ — by Lemma 3.2 over Δ on (4) and exchange over Δ

$\square$

Candidate Expansion Expression Validation

The *candidate expansion expression validation judgement*, $\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \rightsquigarrow e : \tau$, 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:

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

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

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

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),

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

the corresponding candidate expansion expression validation rule is

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

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 $\Delta\ \Gamma \vdash^{\mathbb{E}} \mathcal{C}(e) \leadsto e : \tau$.*

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:

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

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) $e = \mathtt{ap}(e_1;e_2)$	by assumption
(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e}_1 \leadsto e_1 : \mathtt{parr}(\tau_1;\tau)$	by assumption
(4) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e}_2 \leadsto e_2 : \tau_1$	by assumption
(5) $\Delta\ \Gamma \vdash e_1 : \mathtt{parr}(\tau_1;\tau)$	by IH, part 1 on (3)
(6) $\Delta\ \Gamma \vdash e_2 : \tau_1$	by IH on (4)
(7) $\Delta\ \Gamma \vdash \mathtt{ap}(e_1;e_2) : \tau$	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) $\hat{e} = \mathtt{usyntaxue}\{e_{\text{parse}}\}\{\hat{\tau}'\}(\hat{a}.\hat{e}')$	by assumption
(2) $\hat{\Delta} \vdash \hat{\tau}' \leadsto \tau'\ \mathsf{type}$	by assumption
(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi},\hat{a}\leadsto a \hookrightarrow \mathtt{uetsm}(\tau';e_{\text{parse}})} \hat{e}' \leadsto e : \tau$	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) $\mathcal{A} = \mathcal{A}', \hat{a} \leadsto a$	by assumption
(3) $\Psi = \Psi', a \hookrightarrow \mathtt{uetsm}(\tau;\ e_{\text{parse}})$	by assumption
(4) $b \downarrow e_{\text{body}}$	by assumption
(5) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \mathtt{Success} \cdot e_{\text{cand}}$	by assumption
(6) $e_{\text{cand}} \uparrow_{\mathsf{CEExp}} \grave{e}$	by assumption
(7) $\emptyset\ \emptyset \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};b} \grave{e} \leadsto e : \tau$	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 $\hat{\Delta}_{\text{app}} = \langle \mathcal{D}; \Delta_{\text{app}} \rangle$ and $\hat{\Gamma}_{\text{app}} = \langle \mathcal{G}; \Gamma_{\text{app}} \rangle$ and $\hat{\Psi} = \langle \mathcal{A}; \Psi \rangle$.

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

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

Case (3.6b). We have

(1)	$\grave{e} = \texttt{celam}\{\grave{\tau}_1\}(x.\grave{e}')$	by assumption
(2)	$e = \texttt{lam}\{\tau_1\}(x.e')$	by assumption
(3)	$\tau = \texttt{parr}(\tau_1; \tau_2)$	by assumption
(4)	$\Delta \vdash^{\hat{\Delta}_{\text{app}}; b} \grave{\tau}_1 \rightsquigarrow \tau_1$ type	by assumption
(5)	$\Delta\ \Gamma, x : \tau_1 \vdash^{\hat{\Delta}_{\text{app}}; \hat{\Gamma}_{\text{app}}; \hat{\Psi}; b} \grave{e}' \rightsquigarrow e' : \tau_2$	by assumption
(6)	$\Delta \cap \Delta_{\text{app}} = \emptyset$	by assumption
(7)	$\text{dom}(\Gamma) \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset$	by assumption
(8)	$x \notin \text{dom}(\Gamma_{\text{app}})$	by identification convention
(9)	$\text{dom}(\Gamma, x : \tau_1) \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset$	by (7) and (8)
(10)	$\Delta \cup \Delta_{\text{app}} \vdash \tau_1$ type	by Lemma 3.19 on (4)
(11)	$\Delta \cup \Delta_{\text{app}}\ \Gamma, x : \tau_1 \cup \Gamma_{\text{app}} \vdash e' : \tau_2$	by IH, part 2 on (5), (6) and (9)
(12)	$\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}}, x : \tau_1 \vdash e' : \tau_2$	by exchange over Γ_{app} on (11)
(13)	$\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash \texttt{lam}\{\tau_1\}(x.e') : \texttt{parr}(\tau_1; \tau_2)$	by Rule (3.2b) on (10) and (12)

Case (3.6c). We have

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

Case (3.6d). We have

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

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

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:

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

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 $\hat{\Delta}_{\text{app}}\ \hat{\Gamma}_{\text{app}} \vdash_{\hat{\Psi}} \hat{e} \leadsto e : \tau$ 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:

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

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. $\square$

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 2.1. 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. miniVerse$_{\mathbf{UE}}$ 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. 2.2.1, 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:

```
fun read_example_rx(r : Rx) : (string * Rx) option =>
  match r with
    /@name: %ssn/ => Some (name, ssn)
  | _ => 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 6, 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 miniVerse$_{\mathbf{UE}}$ called miniVerse$_{\mathbf{U}}$ 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:

```
fun read_example_rx(r : Rx) : (string * Rx) option =>
  match r with
    $rx /@name: %ssn/ => Some (name, ssn)
  | _ => 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* [18].[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
$\Delta\ \Gamma \vdash r : \tau \Rrightarrow \tau'$	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 miniVerse$_{\mathbf{U}}$ are exactly those of 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:

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

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 $1 \le i \le n$, 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*, $\Delta\ \Gamma \vdash r : \tau \Rrightarrow \tau'$, which is defined mutually with Rules (4.1) by the following rule:

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

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
			$\texttt{in}[L][\ell]\{\{i \hookrightarrow \tau_i\}_{i\in L}\}(e)$	$\ell \cdot e$	injection
			$\texttt{match}[n]\{\tau\}(e;\{r_i\}_{1\leq i\leq n})$	$\texttt{match}\, e\, \{r_i\}_{1\leq i\leq n}$	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
			$\texttt{inp}[\ell](p)$	$\ell \cdot p$	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 $\{r_i\}_{1\leq i\leq n}$ for sequences of $n \geq 0$ rule arguments and $p.e$ for expressions binding the variables that appear in the pattern p. Types and expanded terms are identified up to α-equivalence.

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

$$\frac{}{\Delta \vdash x : \tau \dashv\!\!| \, x : \tau} \tag{4.3a}$$

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

$$\frac{\Delta \vdash p : [\texttt{rec}(t.\tau)/t]\tau \dashv\!\!| \, \Upsilon}{\Delta \vdash \texttt{foldp}(p) : \texttt{rec}(t.\tau) \dashv\!\!| \, \Upsilon} \tag{4.3c}$$

$$\frac{\{\Delta \vdash p_i : \tau_i \dashv\!\!| \, \Upsilon_i\}_{i \in L}}{\Delta \vdash \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\!\!| \, \cup_{i \in L} \Upsilon_i} \tag{4.3d}$$

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

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 Υ if p matches values of a single unrolling of the recursive type, $[\texttt{rec}(t.\tau)/t]\tau$, and generates hypotheses Υ.

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 Υ_i. The conclusion of the rule gathers these hypotheses into a single pattern typing context, $\cup_{i \in L} \Upsilon_i$. The definition of typing context extension, applied iteratively here, implicitly requires that the pattern typing contexts Υ_i be mutually disjoint, i.e.

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

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

2. The final premise of Rule (4.2) extends the typing context, Γ, with the hypotheses generated by pattern typing, Υ, 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* $\Delta \vdash p : \tau \dashv\!\mid \Upsilon$ *then* Δ, t type $\vdash p : \tau \dashv\!\mid \Upsilon$*.*

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* [19], 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 $\Delta\;\Gamma \vdash [e'/x]r : \tau \Mapsto \tau''$ and $\Delta\;\Gamma \vdash e' : \tau'$ then $\Delta\;\Gamma, x : \tau' \vdash r : \tau \Mapsto \tau''$.

Proof Sketch.

1. By rule induction over Rules (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 $\Delta\;[\tau'/t]\Gamma \vdash [\tau'/t]r : [\tau'/t]\tau \Mapsto [\tau'/t]\tau''$ 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 $\Delta\;\Gamma \vdash [e'/x]r : \tau \Mapsto \tau''$ does not depend on the form of e'.

□

The Pattern Regularity Lemma establishes that the hypotheses generated by checking a pattern against a well-formed type involve only well-formed types.

Lemma 4.4 (Pattern Regularity)**.** *If $\Delta \vdash p : \tau \dashv\!\!| \Upsilon$ and $\Delta \vdash \tau$ type then $\Delta \vdash \Upsilon$ ctx.*

Proof. By rule induction over Rules (4.3).

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

(1) $p = x$	by assumption
(2) $\Upsilon = x : \tau$	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) $\Upsilon = \emptyset$	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) $\tau = \mathbf{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})$	by assumption
(3) $\Upsilon = \cup_{i \in L} \Upsilon_i$	by assumption
(4) $\{\Delta \vdash p_i : \tau_i \dashv\!\!| \Upsilon_i\}_{i \in L}$	by assumption
(5) $\Delta \vdash \mathbf{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})$ type	by assumption
(6) $\{\Delta \vdash \tau_i \text{ type}\}_{i \in L}$	by Inversion of Rule (3.1e) on (5)
(7) $\{\Delta \vdash \Upsilon_i \text{ ctx}\}_{i \in L}$	by IH over (4) and (6)

(8) $\Delta \vdash \cup_{i \in L} \Upsilon_i$ 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) $p = \texttt{inp[}\ell\texttt{]}(p')$	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) $\Delta \vdash p' : \tau' \dashV \Upsilon$	by assumption
(5) $\Delta \vdash \tau'$ type	by Inversion of Rule (3.1f) on (3)
(6) $\Delta \vdash \Upsilon$ 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* $\Delta\ \Gamma \vdash r : \tau \Rrightarrow \tau'$ *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* miniVerse$_\mathbf{U}$ 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* $\tau = \texttt{parr(}\tau_1\texttt{;}\tau_2\texttt{)}$ *then* $e = \texttt{lam\{}\tau_1\texttt{\}(}x.e'\texttt{)}$ *and* $x : \tau_1 \vdash e' : \tau_2$.
2. *If* $\tau = \texttt{all(}t.\tau'\texttt{)}$ *then* $e = \texttt{tlam(}t.e'\texttt{)}$ *and* t type $\vdash e' : \tau'$.
3. *If* $\tau = \texttt{rec(}t.\tau'\texttt{)}$ *then* $e = \texttt{fold\{}t.\tau'\texttt{\}(}e'\texttt{)}$ *and* $\vdash e' : [\texttt{rec(}t.\tau'\texttt{)}/t]\tau'$ *and* e' val.
4. *If* $\tau = \texttt{prod}[L]\texttt{(}\{i \hookrightarrow \tau_i\}_{i\in L}\texttt{)}$ *then* $e = \texttt{tpl}[L]\texttt{(}\{i \hookrightarrow e_i\}_{i\in L}\texttt{)}$ *and* $\vdash e_i : \tau_i$ *and* e_i val *for each* $i \in L$.
5. *If* $\tau = \texttt{sum}[L]\texttt{(}\{i \hookrightarrow \tau_i\}_{i\in L}\texttt{)}$ *then for some label set* L' *and label* ℓ *and type* τ_ℓ*, we have that* $L = L', \ell$ *and* $\tau = \texttt{sum}[L', \ell]\texttt{(}\{i \hookrightarrow \tau_i\}_{i\in L'}\texttt{;}\ell \hookrightarrow \tau_\ell\texttt{)}$ *and* $e = \texttt{in}[L', \ell]\texttt{[}\ell\texttt{]\{}\{i \hookrightarrow \tau_i\}_{i\in L'}\texttt{;}\ell \hookrightarrow \tau_\ell\texttt{\}(}e'\texttt{)}$ *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 [18] for a formal account).

4.2.4 Syntax of the Outer Surface

A $\mathsf{miniVerse_U}$ program ultimately evaluates as an expanded expression. However, the programmer does not write the expanded expression directly. Instead, the programmer writes a textual sequence, b, consisting of characters in some suitable alphabet (e.g. in practice, `ASCII` or `Unicode`), which is parsed by some partial metafunction $\mathsf{parseUExp}(b)$ to produce an *unexpanded expression*, $\hat{e}$. Unexpanded expressions can contain *unexpanded types*, $\hat{\tau}$, *unexpanded rules*, $\hat{r}$, and *unexpanded patterns*, $\hat{p}$, so we also need partial metafunctions $\mathsf{parseUTyp}(b)$, $\mathsf{parseURule}(b)$ and $\mathsf{parseUPat}(b)$. The abstract syntax of unexpanded types, expressions, rules and patterns, which form the *outer surface* of $\mathsf{miniVerse_U}$, is defined in Figure 4.3. The full definition of the textual syntax of $\mathsf{miniVerse_U}$ 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* $\mathsf{parseUTyp}(b) = \hat{\tau}$.
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
			$\mathtt{uparr}(\hat{\tau};\hat{\tau})$	$\hat{\tau} \rightharpoonup \hat{\tau}$	partial function
			$\mathtt{uall}(\hat{t}.\hat{\tau})$	$\forall\hat{t}.\hat{\tau}$	polymorphic
			$\mathtt{urec}(\hat{t}.\hat{\tau})$	$\mu\hat{t}.\hat{\tau}$	recursive
			$\mathtt{uprod}[L](\{i \hookrightarrow \hat{\tau}_i\}_{i\in L})$	$\langle\{i \hookrightarrow \hat{\tau}_i\}_{i\in L}\rangle$	labeled product
			$\mathtt{usum}[L](\{i \hookrightarrow \hat{\tau}_i\}_{i\in L})$	$[\{i \hookrightarrow \hat{\tau}_i\}_{i\in L}]$	labeled sum
UExp	$\hat{e}$	$::=$	$\hat{x}$	$\hat{x}$	sigil
			$\mathtt{ulam}\{\hat{\tau}\}(\hat{x}.\hat{e})$	$\lambda\hat{x}{:}\hat{\tau}.\hat{e}$	abstraction
			$\mathtt{uap}(\hat{e};\hat{e})$	$\hat{e}(\hat{e})$	application
			$\mathtt{utlam}(\hat{t}.\hat{e})$	$\Lambda\hat{t}.\hat{e}$	type abstraction
			$\mathtt{utap}\{\hat{\tau}\}(\hat{e})$	$\hat{e}[\hat{\tau}]$	type application
			$\mathtt{ufold}\{\hat{t}.\hat{\tau}\}(\hat{e})$	$\mathtt{fold}(\hat{e})$	fold
			$\mathtt{uunfold}(\hat{e})$	$\mathtt{unfold}(\hat{e})$	unfold
			$\mathtt{utpl}[L](\{i \hookrightarrow \hat{e}_i\}_{i\in L})$	$\langle\{i \hookrightarrow \hat{e}_i\}_{i\in L}\rangle$	labeled tuple
			$\mathtt{upr}[\ell](\hat{e})$	$\hat{e}\cdot\ell$	projection
			$\mathtt{uin}[L][\ell]\{\{i \hookrightarrow \hat{\tau}_i\}_{i\in L}\}(\hat{e})$	$\ell\cdot\hat{e}$	injection
			$\mathtt{umatch}[n]\{\hat{\tau}\}(\hat{e};\{\hat{r}_i\}_{1\leq i\leq n})$	$\mathtt{match}\ \hat{e}\ \{\hat{r}_i\}_{1\leq i\leq n}$	match
			$\mathtt{usyntaxue}\{e\}\{\hat{\tau}\}(\hat{a}.\hat{e})$	$\mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{for}$ $\mathtt{expressions}\ \{e\}\ \mathtt{in}\ \hat{e}$	ueTSM definition
			$\mathtt{uapuetsm}[b][\hat{a}]$	$\hat{a}\ /b/$	ueTSM application
			$\mathtt{usyntaxup}\{e\}\{\hat{\tau}\}(\hat{a}.\hat{e})$	$\mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{for}$ $\mathtt{patterns}\ \{e\}\ \mathtt{in}\ \hat{e}$	upTSM definition
URule	$\hat{r}$	$::=$	$\mathtt{urule}(\hat{p}.\hat{e})$	$\hat{p} \Rightarrow \hat{e}$	match rule
UPat	$\hat{p}$	$::=$	$\hat{x}$	$\hat{x}$	sigil pattern
			$\mathtt{uwildp}$	_	wildcard pattern
			$\mathtt{ufoldp}(\hat{p})$	$\mathtt{fold}(\hat{p})$	fold pattern
			$\mathtt{utplp}[L](\{i \hookrightarrow \hat{p}_i\}_{i\in L})$	$\langle\{i \hookrightarrow \hat{p}_i\}_{i\in L}\rangle$	labeled tuple pattern
			$\mathtt{uinp}[\ell](\hat{p})$	$\ell\cdot\hat{p}$	injection pattern
			$\mathtt{uapuptsm}[b][\hat{a}]$	$\hat{a}\ /b/$	upTSM application

Figure 4.3: Abstract syntax of unexpanded types, expressions, rules and patterns in miniVerse$_{\mathbf{U}}$. 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:

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

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

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

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

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

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

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**
$\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type}$	$\hat{\tau}$ is well-formed and has expansion τ assuming $\hat{\Delta}$
$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e : \tau$	$\hat{e}$ has expansion e and type τ under ueTSM context $\hat{\Psi}$ and upTSM context $\hat{\Phi}$ assuming $\hat{\Delta}$ and $\hat{\Gamma}$
$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \rightsquigarrow r : \tau \Rrightarrow \tau'$	$\hat{r}$ 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*, $\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type}$, 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*, $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \rightsquigarrow r : \tau \Rrightarrow \tau'$ 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:

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

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

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

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

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

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).

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

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

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

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.

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

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.

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

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

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

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

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

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

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

and

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

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 $\Delta \vdash p : \tau \dashv\!\!| \Upsilon$ then $\Delta \vdash_{\hat{\Phi}} \mathcal{U}(p) \rightsquigarrow p : \tau \dashv\!\!| \mathcal{U}(\Upsilon)$.*

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

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

For part 1, we induct on the assumption. 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)	$\mathcal{U}(\Delta)\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{urule}(\mathcal{U}(p).\mathcal{U}(e)) \rightsquigarrow \texttt{rule}(p.e) : \tau \Mapsto \tau'$	by Rule (4.5) on (8) and (9)

$\square$

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

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

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

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 $a \in \text{dom}(\Psi)$ to an *expanded ueTSM definition*, $a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}})$, where τ is the ueTSM's type annotation, and e_{parse} is its parse function. We write $\Psi, a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}})$ 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 4.14 (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* $\emptyset\ \emptyset \vdash e_{parse} : \texttt{Body} \rightharpoonup \texttt{ParseResultExp}$.

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 type abbreviated $\texttt{Body}$ classifies encodings of literal bodies, b. The mapping from literal bodies to values of type $\texttt{Body}$ 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 $\texttt{Body}$ 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 $b = b'$.*

$\texttt{ParseResultExp}$ 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 $\texttt{CEExp}$ 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 $\texttt{CEExp}$ 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 $\texttt{CEExp}$ 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:

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

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*, $\grave{p} \downarrow_{\mathsf{CEPat}} e$. An inverse mapping is defined by the *ce-pattern decoding judgement*, $e \uparrow_{\mathsf{CEPat}} \grave{p}$.

Judgement Form	**Description**
$\grave{p} \downarrow_{\mathsf{CEPat}} e$	$\grave{p}$ has encoding e
$e \uparrow_{\mathsf{CEPat}} \grave{p}$	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 $\grave{p} \downarrow_{\mathsf{CEPat}} e_{cand}$ 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* $e_{cand} \uparrow_{\mathsf{CEPat}} \grave{p}$ *for some* $\grave{p}$.

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

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

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

(f) *If* $\vdash e_{cand} : \texttt{CEPat}$ *and* e_{cand} val *and* $e_{cand} \uparrow_{\mathsf{CEPat}} \grave{p}$ *and* $e_{cand} \uparrow_{\mathsf{CEPat}} \grave{p}'$ *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 $a \in \text{dom}(\Phi)$ 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 $a \notin \text{dom}(\Phi)$ 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* $\emptyset\, \emptyset \vdash e_{parse} : \texttt{Body} \rightharpoonup \texttt{ParseResultPat}$.

   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 $\texttt{uapuptsm}[b][\hat{a}]$.

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

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

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 $\texttt{Success} \cdot e_{\text{cand}}$ results from evaluation, then e_{cand} will be a value of type $\texttt{CEPat}$ (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:

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

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

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

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
			$\texttt{cesplicedt}[m;n]$	$\texttt{spliced}\langle m,n\rangle$	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
			$\texttt{cein}[L][\ell]\{\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}\}(\grave{e})$	$\ell \cdot \grave{e}$	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
			$\texttt{cesplicede}[m;n]$	$\texttt{spliced}\langle m,n\rangle$	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
			$\texttt{ceinp}[\ell](\grave{p})$	$\ell \cdot \grave{p}$	injection pattern
			$\texttt{cesplicedp}[m;n]$	$\texttt{spliced}\langle m,n\rangle$	spliced

Figure 4.4: Abstract syntax of candidate expansion types, expressions, rules and patterns in miniVerse$_\mathbf{U}$. 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:

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

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 $\mathbb{E}$
$\vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : \tau \dashv\mid^{\hat{\Upsilon}}$	$\grave{p}$ expands to p and matches values of type τ generating assumptions $\hat{\Upsilon}$ 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:

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

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

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

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

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

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

(4) $\Delta \vdash \tau$ type — by assumption

(5) $\{\Delta\, \Gamma \vdash r_i : \tau' \Mapsto \tau\}_{1 \le i \le n}$ — 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) $\{\Delta\, \Gamma \vdash^{\mathbb{E}} \mathcal{C}(r_i) \rightsquigarrow r_i : \tau' \Mapsto \tau\}_{1 \le i \le n}$ — by IH, part 2 over (5)

(9) $\Delta\, \Gamma \vdash^{\mathbb{E}} \mathtt{cematch}[n]\{\mathcal{C}(\tau)\}(\mathcal{C}(e'); \{\mathcal{C}(r_i)\}_{1 \le i \le n}) \rightsquigarrow \mathtt{match}[n]\{\tau\}(e'; \{r_i\}_{1 \le i \le n}) : \tau$ — 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) $\Delta \vdash p : \tau \dashv\!\!| \Upsilon$ — by assumption

(4) $\Delta\, \Gamma \cup \Upsilon \vdash e : \tau'$ — by assumption

(5) $\Delta\, \Gamma \cup \Upsilon \vdash^{\mathbb{E}} \mathcal{C}(e) \rightsquigarrow e : \tau'$ — by IH, part 1 on (4)

(6) $\Delta\, \Gamma \vdash^{\mathbb{E}} \mathtt{cerule}(p.\mathcal{C}(e)) \rightsquigarrow \mathtt{rule}(p.e) : \tau \Mapsto \tau'$ — 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*, $\vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : \tau \dashv\!\!|^{\hat{\Upsilon}}$, 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 $\hat{\Upsilon}$. Hypotheses can be generated only by spliced subpatterns, so there is no ce-pattern form corresponding to variable patterns (this is also why $\hat{\Upsilon}$ 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.

$$\frac{}{\vdash^{\mathbb{P}} \mathtt{cewildp} \rightsquigarrow \mathtt{wildp} : \tau \dashv\!\!|^{\langle \emptyset; \emptyset \rangle}} \quad (4.9a)$$

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

$$\frac{\{\vdash^{\mathbb{P}} \grave{p}_i \rightsquigarrow p_i : \tau_i \dashV^{\hat{\Upsilon}_i}\}_{i \in L}}{\left(\begin{array}{c}\vdash^{\mathbb{P}} \texttt{cetplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i \in L}) \\ \rightsquigarrow \\ \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashV^{\cup_{i \in L} \hat{\Upsilon}_i}\end{array}\right)} \tag{4.9c}$$

$$\frac{\vdash^{\mathbb{P}} \grave{p} \rightsquigarrow p : \tau \dashV^{\hat{\Upsilon}}}{\vdash^{\mathbb{P}} \texttt{ceinp}[\ell](\grave{p}) \rightsquigarrow \texttt{inp}[\ell](p) : \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashV^{\hat{\Upsilon}}} \tag{4.9d}$$

$$\frac{\mathsf{parseUPat}(\mathsf{subseq}(b; m; n)) = \hat{p} \qquad \Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashV \hat{\Upsilon}}{\vdash^{\Delta; \hat{\Phi}; b} \texttt{cesplicedp}[m; n] \rightsquigarrow p : \tau \dashV^{\hat{\Upsilon}}} \tag{4.9e}$$

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 Υ. 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* $\Delta \vdash p : \tau \dashV \emptyset$ *then* $\vdash^{\Delta; \hat{\Phi}; b} \mathcal{C}(p) \rightsquigarrow p : \tau \dashV^{\langle \emptyset; \emptyset \rangle}$.

Proof. By rule induction over Rules (4.3).

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

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

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

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

(1) $p = \texttt{foldp}(p')$	by assumption
(2) $\mathcal{C}(p) = \texttt{cefoldp}(\mathcal{C}(p'))$	by definition of $\mathcal{C}(p)$
(3) $\tau = \texttt{rec}(t.\tau')$	by assumption
(4) $\Delta \vdash p' : [\texttt{rec}(t.\tau')/t]\tau' \dashV \emptyset$	by assumption
(5) $\vdash^{\Delta; \hat{\Phi}; b} \mathcal{C}(p') \rightsquigarrow p : [\texttt{rec}(t.\tau')/t]\tau' \dashV^{\langle \emptyset; \emptyset \rangle}$	by IH on (4)
(6) $\vdash^{\Delta; \hat{\Phi}; b} \texttt{cefoldp}(\mathcal{C}(p')) \rightsquigarrow \texttt{foldp}(p') : \texttt{rec}(t.\tau') \dashV^{\langle \emptyset; \emptyset \rangle}$	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) $\{\Delta \vdash p_i : \tau_i \dashv\!\vert\, \emptyset\}_{i \in L}$ by assumption
(5) $\{\vdash^{\Delta;\hat{\Phi};b} \mathcal{C}(p_i) \rightsquigarrow p_i : \tau_i \dashv\!\vert^{\langle\emptyset;\emptyset\rangle}\}_{i \in L}$ by IH over (4)
(6) $\vdash^{\Delta;\hat{\Phi};b} \texttt{cetpl}[L](\{i \hookrightarrow \mathcal{C}(p_i)\}_{i \in L}) \rightsquigarrow \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) :$
$\texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\!\vert^{\langle\emptyset;\emptyset\rangle}$ 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) $\Delta \vdash p' : \tau' \dashv\!\vert\, \emptyset$ by assumption
(5) $\vdash^{\Delta;\hat{\Phi};b} \mathcal{C}(p') \rightsquigarrow p' : \tau' \dashv\!\vert^{\langle\emptyset;\emptyset\rangle}$ by IH on (4)
(6) $\vdash^{\Delta;\hat{\Phi};b} \texttt{ceinp}[\ell](\mathcal{C}(p')) \rightsquigarrow \texttt{inp}[\ell](p') : \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow$
$\tau') \dashv\!\vert^{\langle\emptyset;\emptyset\rangle}$ 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* $\Delta \vdash_{\langle \mathcal{A}; \Phi \rangle} \hat{p} \rightsquigarrow p : \tau \dashv\!\vert\, \langle \mathcal{G}; \Upsilon \rangle$ *then* $\Delta \vdash p : \tau \dashv\!\vert\, \Upsilon$.
2. *If* $\vdash^{\Delta; \langle \mathcal{A}; \Phi \rangle; b} \grave{p} \rightsquigarrow p : \tau \dashv\!\vert^{\langle \mathcal{G}; \Upsilon \rangle}$ *then* $\Delta \vdash p : \tau \dashv\!\vert\, \Upsilon$.

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) $\Delta \vdash x : \tau \dashv\!\vert\, x : \tau$ by Rule (4.3a)

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

   (1) $p = \texttt{wildp}$ by assumption
   (2) $\Upsilon = \emptyset$ by assumption
   (3) $\Delta \vdash \texttt{wildp} : \tau \dashv\!\vert\, \emptyset$ 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) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{Success} \cdot e_{\text{cand}}$ by assumption

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

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

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

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

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

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

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

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

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

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

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

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:

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

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 $\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \hat{\Upsilon}$. Because the metric is stable when passing from part 1 to part 2, we must have that it is strictly decreasing in the other direction:

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

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 $\langle\mathcal{D};\Delta\rangle\ \langle\mathcal{G};\Gamma\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \rightsquigarrow r : \tau \Rightarrow \tau'$ *then* $\Delta\ \Gamma \vdash r : \tau \Rightarrow \tau'$.
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 $\Delta\ \Gamma \vdash^{\langle\mathcal{D};\Delta_{app}\rangle;\langle\mathcal{G};\Gamma_{app}\rangle;\hat{\Psi};\hat{\Phi};b} \grave{r} \rightsquigarrow r : \tau \Rightarrow \tau'$ *and* $\Delta \cap \Delta_{app} = \emptyset$ *and* $dom(\Gamma) \cap dom(\Gamma_{app}) = \emptyset$ *then* $\Delta \cup \Delta_{app}\ \Gamma \cup \Gamma_{app} \vdash r : \tau \Rightarrow \tau'$.

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

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

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

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

(1)	$\hat{e} = \mathtt{uapuetsm}[b][\hat{a}]$	by assumption
(2)	$\hat{\Psi} = \hat{\Psi}', \hat{a} \rightsquigarrow a \hookrightarrow \mathtt{uetsm}(\tau; e_{\text{parse}})$	by assumption
(3)	$b \downarrow e_{\text{body}}$	by assumption
(4)	$e_{\text{parse}}(e_{\text{body}}) \Downarrow \mathtt{Success} \cdot e_{\text{cand}}$	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) $\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\!\mid \langle \mathcal{A}'; \Upsilon \rangle$ by assumption

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

(5) $\Delta \vdash p : \tau \dashv\!\!\mid \Upsilon$ by Theorem 4.21, part 1 on (3)

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

(7) $\Delta\ \Gamma \vdash \texttt{rule}(p.e) : \tau \Rrightarrow \tau'$ 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) $\{\Delta\ \Gamma \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{r}_i \rightsquigarrow r_i : \tau' \Rrightarrow \tau\}_{1\leq i\leq n}$ 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) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash r : \tau' \Rrightarrow \tau$ 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) $\grave{r} = \mathtt{cerule}(p.\grave{e})$	by assumption
(2) $r = \mathtt{rule}(p.e)$	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) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash \mathtt{rule}(p.e) : \tau \Mapsto \tau'$	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 $\hat{e}$,

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

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

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

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:

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

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. $\square$

Chapter 5

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.

5.1 TSM Implicits By Example

We begin in this section by introducing TSM implicits by example in VerseML. In Sec. 5.2, we formalize unparameterized TSM implicits with a reduced calculus, $\text{miniVerse}_{\mathbf{U}}^{\mathbf{B}}$. We will also return to the topic of TSM implicits after introducing parameterized TSMs in Chapter 6.

5.1.1 Designation

In the example in Figure 5.1, 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 5.1:

```
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 5.1: An example of TSM implicits in VerseML

5.1.2 Usage

On Line 5 of Figure 5.1, 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.

5.1.3 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.

5.2 $\mathsf{miniVerse}^{\mathbf{B}}_{\mathbf{U}}$

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

5.2.1 Inner Core

The inner core of $\mathsf{miniVerse}^{\mathbf{B}}_{\mathbf{U}}$ is the same as the inner core of $\mathsf{miniVerse}_{\mathbf{U}}$, 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.

5.2.2 Syntax of the Outer Surface

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

Condition 5.1 (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
	$\mathtt{uparr}(\hat{\tau};\hat{\tau})$	$\hat{\tau} \rightharpoonup \hat{\tau}$	partial function
	$\mathtt{uall}(\hat{t}.\hat{\tau})$	$\forall\hat{t}.\hat{\tau}$	polymorphic
	$\mathtt{urec}(\hat{t}.\hat{\tau})$	$\mu\hat{t}.\hat{\tau}$	recursive
	$\mathtt{uprod}[L](\{i \hookrightarrow \hat{\tau}_i\}_{i\in L})$	$\langle\{i \hookrightarrow \hat{\tau}_i\}_{i\in L}\rangle$	labeled product
	$\mathtt{usum}[L](\{i \hookrightarrow \hat{\tau}_i\}_{i\in L})$	$[\{i \hookrightarrow \hat{\tau}_i\}_{i\in L}]$	labeled sum
UExp $\hat{e}$ ::=	$\hat{x}$	$\hat{x}$	sigil
	$\mathtt{uasc}\{\hat{\tau}\}(\hat{e})$	$\hat{e} : \hat{\tau}$	ascription
	$\mathtt{uletval}(\hat{e};\hat{x}.\hat{e})$	$\mathtt{let}\ \mathtt{val}\ \hat{x} = \hat{e}\ \mathtt{in}\ \hat{e}$	value binding
	$\mathtt{uanalam}(\hat{x}.\hat{e})$	$\lambda\hat{x}.\hat{e}$	abstraction (unannotated)
	$\mathtt{ulam}\{\hat{\tau}\}(\hat{x}.\hat{e})$	$\lambda\hat{x}{:}\hat{\tau}.\hat{e}$	abstraction (annotated)
	$\mathtt{uap}(\hat{e};\hat{e})$	$\hat{e}(\hat{e})$	application
	$\mathtt{utlam}(\hat{t}.\hat{e})$	$\Lambda\hat{t}.\hat{e}$	type abstraction
	$\mathtt{utap}\{\hat{\tau}\}(\hat{e})$	$\hat{e}[\hat{\tau}]$	type application
	$\mathtt{ufold}(\hat{e})$	$\mathtt{fold}(\hat{e})$	fold
	$\mathtt{uunfold}(\hat{e})$	$\mathtt{unfold}(\hat{e})$	unfold
	$\mathtt{utpl}[L](\{i \hookrightarrow \hat{e}_i\}_{i\in L})$	$\langle\{i \hookrightarrow \hat{e}_i\}_{i\in L}\rangle$	labeled tuple
	$\mathtt{upr}[\ell](\hat{e})$	$\hat{e}\cdot\ell$	projection
	$\mathtt{uin}[\ell](\hat{e})$	$\ell\cdot\hat{e}$	injection
	$\mathtt{umatch}[n](\hat{e};\{\hat{r}_i\}_{1\leq i\leq n})$	$\mathtt{match}\ \hat{e}\ \{\hat{r}_i\}_{1\leq i\leq n}$	match
	$\mathtt{usyntaxue}\{e\}\{\hat{\tau}\}(\hat{a}.\hat{e})$	$\mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{for}$ $\mathtt{expressions}\ \{e\}\ \mathtt{in}\ \hat{e}$	ueTSM definition
	$\mathtt{uimplicite}[\hat{a}](\hat{e})$	$\mathtt{implicit}\ \mathtt{syntax}\ \hat{a}\ \mathtt{for}$ $\mathtt{expressions}\ \mathtt{in}\ \hat{e}$	ueTSM designation
	$\mathtt{uapuetsm}[b][\hat{a}]$	$\hat{a}\ \mathtt{/}b\mathtt{/}$	ueTSM application
	$\mathtt{uelit}[b]$	$\mathtt{/}b\mathtt{/}$	ueTSM unadorned literal
	$\mathtt{usyntaxup}\{e\}\{\hat{\tau}\}(\hat{a}.\hat{e})$	$\mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\tau}\ \mathtt{for}$ $\mathtt{patterns}\ \{e\}\ \mathtt{in}\ \hat{e}$	upTSM definition
	$\mathtt{uimplicitp}[\hat{a}](\hat{e})$	$\mathtt{implicit}\ \mathtt{syntax}\ \hat{a}\ \mathtt{for}$ $\mathtt{patterns}\ \mathtt{in}\ \hat{e}$	upTSM designation
URule $\hat{r}$ ::=	$\mathtt{urule}(\hat{p}.\hat{e})$	$\hat{p} \Rightarrow \hat{e}$	match rule
UPat $\hat{p}$ ::=	$\hat{x}$	$\hat{x}$	sigil pattern
	$\mathtt{uwildp}$	$\_$	wildcard pattern
	$\mathtt{ufoldp}(\hat{p})$	$\mathtt{fold}(\hat{p})$	fold pattern
	$\mathtt{utplp}[L](\{i \hookrightarrow \hat{p}_i\}_{i\in L})$	$\langle\{i \hookrightarrow \hat{p}_i\}_{i\in L}\rangle$	labeled tuple pattern
	$\mathtt{uinp}[\ell](\hat{p})$	$\ell\cdot\hat{p}$	injection pattern
	$\mathtt{uapuptsm}[b][\hat{a}]$	$\hat{a}\ \mathtt{/}b\mathtt{/}$	upTSM application
	$\mathtt{uplit}[b]$	$\mathtt{/}b\mathtt{/}$	upTSM unadorned literal

Figure 5.2: 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:

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

  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:

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

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

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

The forms related to TSM implicits are highlighted in gray in Figure 5.2.

5.2.3 Bidirectionally Typed Expansion

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

Judgement Form	**Description**
$\hat{\Delta} \vdash \hat{\tau} \leadsto \tau\ \mathsf{type}$	$\hat{\tau}$ is well-formed and has expansion τ assuming $\hat{\Delta}$
$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau$	$\hat{e}$ has expansion e and synthesizes type τ under $\hat{\Psi}$ and $\hat{\Phi}$ assuming $\hat{\Delta}$ and $\hat{\Gamma}$
$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Leftarrow \tau$	$\hat{e}$ has expansion e when analyzed against type τ under $\hat{\Psi}$ and $\hat{\Phi}$ assuming $\hat{\Delta}$ and $\hat{\Gamma}$
$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \leadsto r \Rightarrow \tau \Mapsto \tau'$	$\hat{r}$ 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 $\hat{\Gamma}$
$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r} \leadsto r \Leftarrow \tau \Mapsto \tau'$	$\hat{r}$ has expansion r and takes values of type τ to values of type τ' when $\tau's$ is provided for analysis under $\hat{\Psi}$ and $\hat{\Phi}$ assuming $\hat{\Delta}$ and $\hat{\Gamma}$
$\Delta \vdash_{\hat{\Phi}} \hat{p} \leadsto p : \tau \dashv\!\!\mid \hat{\Upsilon}$	$\hat{p}$ has expansion p and type τ and generates hypotheses $\hat{\Upsilon}$ 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*, $\hat{\Delta} \vdash \hat{\tau} \leadsto \tau\ \mathsf{type}$, 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*, $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau$, for type synthesis, and $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Leftarrow \tau$, for type analysis, are defined mutually inductively by Rules (5.1) and Rules (5.2), 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.

$$\frac{}{\hat{\Delta}\ \hat{\Gamma}, \hat{x} \leadsto x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{x} \leadsto x \Rightarrow \tau} \tag{5.1a}$$

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.

$$\frac{\hat{\Delta} \vdash \hat{\tau} \leadsto \tau\ \mathsf{type} \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Leftarrow \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \mathtt{uasc}\{\hat{\tau}\}(\hat{e}) \leadsto e \Rightarrow \tau} \tag{5.1b}$$

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

$$\frac{\hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau \qquad \hat{\Delta}\;\hat{\Gamma}, \hat{x} \leadsto x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \leadsto e' \Rightarrow \tau'}{\hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{uletval}(\hat{e};\hat{x}.\hat{e}') \leadsto \texttt{ap}(\texttt{lam}\{\tau\}(x.e');e) \Rightarrow \tau'} \tag{5.1c}$$

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

$$\frac{\hat{\Delta} \vdash \hat{\tau}_1 \leadsto \tau_1 \;\mathsf{type} \qquad \hat{\Delta}\;\hat{\Gamma}, \hat{x} \leadsto x : \tau_1 \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau_2}{\hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{ulam}\{\hat{\tau}_1\}(\hat{x}.\hat{e}) \leadsto \texttt{lam}\{\tau_1\}(x.e) \Rightarrow \texttt{parr}(\tau_1;\tau_2)} \tag{5.1d}$$

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

$$\frac{\hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_1 \leadsto e_1 \Rightarrow \texttt{parr}(\tau_2;\tau) \qquad \hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_2 \leadsto e_2 \Leftarrow \tau_2}{\hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{uap}(\hat{e}_1;\hat{e}_2) \leadsto \texttt{ap}(e_1;e_2) \Rightarrow \tau} \tag{5.1e}$$

Type lambdas and type applications can appear in synthetic position.

$$\frac{\hat{\Delta}, \hat{t} \leadsto t \;\mathsf{type}\; \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau}{\hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{utlam}(\hat{t}.\hat{e}) \leadsto \texttt{tlam}(t.e) \Rightarrow \texttt{all}(t.\tau)} \tag{5.1f}$$

$$\frac{\hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \texttt{all}(t.\tau) \qquad \hat{\Delta} \vdash \hat{\tau}' \leadsto \tau' \;\mathsf{type}}{\hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{utap}\{\hat{\tau}'\}(\hat{e}) \leadsto \texttt{tap}\{\tau'\}(e) \Rightarrow [\tau'/t]\tau} \tag{5.1g}$$

Unfoldings can appear in synthetic position.

$$\frac{\hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \texttt{rec}(t.\tau)}{\hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{uunfold}(\hat{e}) \leadsto \texttt{unfold}(e) \Rightarrow [\texttt{rec}(t.\tau)/t]\tau} \tag{5.1h}$$

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

$$\frac{\{\hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_i \leadsto e_i \Rightarrow \tau_i\}_{i\in L}}{\hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{utpl}[L](\{i \hookrightarrow \hat{e}_i\}_{i\in L}) \leadsto \texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L}) \Rightarrow \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})} \tag{5.1i}$$

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

$$\frac{\hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \texttt{prod}[L,\ell](\{i \hookrightarrow \tau_i\}_{i\in L};\ell \hookrightarrow \tau)}{\hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{upr}[\ell](\hat{e}) \leadsto \texttt{pr}[\ell](e) \Rightarrow \tau} \tag{5.1j}$$

Match expressions can appear in synthetic position.

$$\frac{\hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau \qquad \{\hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \leadsto r_i \Rightarrow \tau \Mapsto \tau'\}_{1\leq i\leq n}}{\hat{\Delta}\;\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{umatch}[n](\hat{e};\{\hat{r}_i\}_{1\leq i\leq n}) \leadsto \texttt{match}[n]\{\tau'\}(e;\{r_i\}_{1\leq i\leq n}) \Rightarrow \tau'} \tag{5.1k}$$

ueTSMs can be defined and applied in synthetic position.

$$\frac{\hat{\Delta} \vdash \hat{\tau} \leadsto \tau \text{ type} \qquad \emptyset\,\emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResultExp)} \qquad \hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi},\hat{a}\leadsto a \hookrightarrow \texttt{uetsm}(\tau;e_{\text{parse}});\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau'}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for expressions}\ \{e_{\text{parse}}\}\ \texttt{in}\ \hat{e} \leadsto e \Rightarrow \tau'} \tag{5.1l}$$

$$\frac{b \downarrow e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{Success} \cdot e_{\text{cand}} \qquad e_{\text{cand}} \uparrow_{\text{CEExp}} \grave{e} \qquad \emptyset\,\emptyset \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi},\hat{a}\leadsto a\hookrightarrow\texttt{uetsm}(\tau;e_{\text{parse}});\hat{\Phi};b} \grave{e} \leadsto e \Leftarrow \tau}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi},\hat{a}\leadsto a\hookrightarrow\texttt{uetsm}(\tau;e_{\text{parse}});\hat{\Phi}} \hat{a}\ /b/ \leadsto e \Rightarrow \tau} \tag{5.1m}$$

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. 5.2.5 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 $\hat{\Psi}, \hat{a} \leadsto a \hookrightarrow \texttt{uetsm}(\tau;\ e_{\text{parse}})$ when $\hat{\Psi} = \langle \mathcal{A};\Psi;\mathcal{I}\rangle$ as shorthand for

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

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:

$$\frac{\hat{\Delta}\,\hat{\Gamma} \vdash_{\langle \mathcal{A}\uplus\hat{a}\leadsto a;\Psi,a\hookrightarrow\texttt{uetsm}(\tau;e_{\text{parse}});\mathcal{I}\uplus\tau\hookrightarrow a\rangle;\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau'}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\langle \mathcal{A}\uplus\hat{a}\leadsto a;\Psi,a\hookrightarrow\texttt{uetsm}(\tau;e_{\text{parse}});\mathcal{I}\rangle;\hat{\Phi}} \texttt{implicit syntax}\ \hat{a}\ \texttt{for expressions in}\ \hat{e} \leadsto e \Rightarrow \tau'} \tag{5.1n}$$

Like ueTSMs, upTSMs can be defined in synthetic position.

$$\frac{\hat{\Delta} \vdash \hat{\tau} \leadsto \tau \text{ type} \qquad \emptyset\,\emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResultPat)} \qquad \hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi},\hat{a}\leadsto a \hookrightarrow \texttt{uptsm}(\tau;e_{\text{parse}})} \hat{e} \leadsto e \Rightarrow \tau'}{\hat{\Delta}\,\hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for patterns}\ \{e_{\text{parse}}\}\ \texttt{in}\ \hat{e} \leadsto e \Rightarrow \tau'} \tag{5.1o}$$

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:

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{uptsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle} \hat{e} \rightsquigarrow e \Rightarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}}); \mathcal{I} \rangle} \texttt{implicit syntax}\ \hat{a}\ \texttt{for patterns in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \tag{5.1p}$$

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.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau} \tag{5.2a}$$

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 (5.1c) governed value bindings in synthetic position. The following rule governs value bindings in analytic position.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e}' \rightsquigarrow e' \Leftarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{uletval}(\hat{e}; \hat{x}.\hat{e}') \rightsquigarrow \texttt{ap}(\texttt{lam}\{\tau\}(x.e'); e) \Leftarrow \tau'} \tag{5.2b}$$

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.

$$\frac{\hat{\Delta}\ \hat{\Gamma}, \hat{x} \rightsquigarrow x : \tau_1 \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau_2}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{uanalam}(\hat{x}.\hat{e}) \rightsquigarrow \texttt{lam}\{\tau_1\}(x.e) \Leftarrow \texttt{parr}(\tau_1; \tau_2)} \tag{5.2c}$$

Rule (5.1f) governed type lambdas in synthetic position. The following rule governs type lambdas in analytic position.

$$\frac{\hat{\Delta}, \hat{t} \rightsquigarrow t\ \mathsf{type}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{utlam}(\hat{t}.\hat{e}) \rightsquigarrow \texttt{tlam}(t.e) \Leftarrow \texttt{all}(t.\tau)} \tag{5.2d}$$

Values of recursive types can be introduced only in analytic position.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow [\texttt{rec}(t.\tau)/t]\tau}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{ufold}(\hat{e}) \rightsquigarrow \texttt{fold}\{t.\tau\}(e) \Leftarrow \texttt{rec}(t.\tau)} \tag{5.2e}$$

Rule (5.1i) governed labeled tuples in synthetic position. The following rule governs labeled tuples in analytic position.

$$\frac{\{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_i \leadsto e_i \Leftarrow \tau_i\}_{i\in L}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{utpl}[L](\{i \hookrightarrow \hat{e}_i\}_{i\in L}) \leadsto \texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L}) \Leftarrow \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})} \tag{5.2f}$$

Values of labeled sum type can appear only in analytic position.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Leftarrow \tau}{\left(\begin{array}{c}\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{uin}[\ell](\hat{e}) \\ \leadsto \\ \texttt{in}[L,\ell][\ell]\{\{i \hookrightarrow \tau_i\}_{i\in L}; \ell \hookrightarrow \tau\}(e) \Leftarrow \texttt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i\in L}; \ell \hookrightarrow \tau)\end{array}\right)} \tag{5.2g}$$

Rule (5.1k) governed match expressions in synthetic position. The following rule governs match expressions in analytic position.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau \qquad \{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \leadsto r_i \Leftarrow \tau \Mapsto \tau'\}_{1\leq i\leq n}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{umatch}[n](\hat{e}; \{\hat{r}_i\}_{1\leq i\leq n}) \leadsto \texttt{match}[n]\{\tau'\}(e; \{r_i\}_{1\leq i\leq n}) \Leftarrow \tau'} \tag{5.2h}$$

Rule (5.1l) governed ueTSM definitions in synthetic position. The following rule governs ueTSM definitions in analytic position.

$$\frac{\begin{array}{c}\hat{\Delta} \vdash \hat{\tau} \leadsto \tau \text{ type} \qquad \emptyset\, \emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResultExp)} \\ \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi},\hat{a}\leadsto a\hookrightarrow\texttt{uetsm}(\tau;e_{\text{parse}});\hat{\Phi}} \hat{e} \leadsto e \Leftarrow \tau'\end{array}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for expressions}\ \{e_{\text{parse}}\}\ \texttt{in}\ \hat{e} \leadsto e \Leftarrow \tau'} \tag{5.2i}$$

Rule (5.1n) governed ueTSM designations in synthetic position. The following rule governs ueTSM designations in analytic position.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle\mathcal{A}\uplus\hat{a}\leadsto a;\Psi,a\hookrightarrow\texttt{uetsm}(\tau;e_{\text{parse}});\mathcal{I}\uplus\tau\hookrightarrow a\rangle;\hat{\Phi}} \hat{e} \leadsto e \Leftarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle\mathcal{A}\uplus\hat{a}\leadsto a;\Psi,a\hookrightarrow\texttt{uetsm}(\tau;e_{\text{parse}});\mathcal{I}\rangle;\hat{\Phi}} \texttt{implicit syntax}\ \hat{a}\ \texttt{for expressions in}\ \hat{e} \leadsto e \Leftarrow \tau'} \tag{5.2j}$$

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 (5.1m).

$$\frac{\begin{array}{c}b \downarrow e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{Success} \cdot e_{\text{cand}} \qquad e_{\text{cand}} \uparrow_{\texttt{CEExp}} \grave{e} \\ \emptyset\, \emptyset \vdash^{\hat{\Delta};\hat{\Gamma};\langle\mathcal{A};\Psi,a\hookrightarrow\texttt{uetsm}(\tau;e_{\text{parse}});\mathcal{I}\uplus\tau\hookrightarrow a\rangle;\hat{\Phi};b} \grave{e} \leadsto e \Leftarrow \tau\end{array}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle\mathcal{A};\Psi,a\hookrightarrow\texttt{uetsm}(\tau;e_{\text{parse}});\mathcal{I}\uplus\tau\hookrightarrow a\rangle;\hat{\Phi}} \texttt{uelit}[b] \leadsto e \Leftarrow \tau} \tag{5.2k}$$

Rule (5.1o) governed upTSM definitions in synthetic position. The following rule governs upTSM definitions in analytic position.

$$\frac{\begin{array}{c}\hat{\Delta} \vdash \hat{\tau} \leadsto \tau \text{ type} \qquad \emptyset\, \emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResultPat)} \\ \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi},\hat{a}\leadsto a\hookrightarrow\texttt{uptsm}(\tau;e_{\text{parse}})} \hat{e} \leadsto e \Leftarrow \tau'\end{array}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for patterns}\ \{e_{\text{parse}}\}\ \texttt{in}\ \hat{e} \leadsto e \Leftarrow \tau'} \tag{5.2l}$$

Rule (5.1p) governed upTSM designations in synthetic position. The following rule governs upTSM designations in analytic position.

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{uptsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}}); \mathcal{I} \rangle} \texttt{implicit syntax}\ \hat{a}\ \texttt{for patterns in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{5.2m}$$

Typed Rule Expansion

The synthetic typed rule expansion judgement is invoked iteratively by Rule (5.1k) to synthesize a type, τ', from the branch expressions in the rule sequence. This judgement is defined mutually inductively with Rules (5.1) and Rules (5.2) by the following rule.

$$\frac{\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\vert\ \langle \mathcal{G}'; \Gamma' \rangle \qquad \langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G} \uplus \mathcal{G}'; \Gamma \cup \Gamma' \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau'}{\langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{urule}(\hat{p}.\hat{e}) \rightsquigarrow \texttt{rule}(p.e) \Rightarrow \tau \Mapsto \tau'} \tag{5.3}$$

The analytic typed rule expansion judgement is invoked iteratively by Rule (5.2h). This judgement is defined mutually inductively with Rules (5.1), Rules (5.2), and Rule (5.3) by the following rule, which is the analytic analag of Rule (5.3).

$$\frac{\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\vert\ \langle \mathcal{G}'; \Gamma' \rangle \qquad \langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G} \uplus \mathcal{G}'; \Gamma \cup \Gamma' \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{urule}(\hat{p}.\hat{e}) \rightsquigarrow \texttt{rule}(p.e) \Leftarrow \tau \Mapsto \tau'} \tag{5.4}$$

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 (5.5) as follows.

The following rules are written identically to the typed pattern expansion rules for shared pattern forms in miniVerse$_{\mathbf{U}}$, i.e. Rules (4.6a) through (4.6e).

$$\frac{}{\Delta \vdash_{\hat{\Phi}} \hat{x} \rightsquigarrow x : \tau \dashv\!\vert\ \langle \hat{x} \rightsquigarrow x; x : \tau \rangle} \tag{5.5a}$$

$$\frac{}{\Delta \vdash_{\hat{\Phi}} \texttt{uwildp} \rightsquigarrow \texttt{wildp} : \tau \dashv\!\vert\ \langle \emptyset; \emptyset \rangle} \tag{5.5b}$$

$$\frac{\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : [\texttt{rec}(t.\tau)/t]\tau \dashv\!\vert\ \hat{\Upsilon}}{\Delta \vdash_{\hat{\Phi}} \texttt{ufoldp}(\hat{p}) \rightsquigarrow \texttt{foldp}(p) : \texttt{rec}(t.\tau) \dashv\!\vert\ \hat{\Upsilon}} \tag{5.5c}$$

$$\frac{\{\Delta \vdash_{\hat{\Phi}} \hat{p}_i \rightsquigarrow p_i : \tau_i \dashv\!\vert\ \hat{\Upsilon}_i\}_{i \in L}}{\left(\begin{array}{c}\Delta \vdash_{\hat{\Phi}} \texttt{utplp}[L](\{i \hookrightarrow \hat{p}_i\}_{i \in L}) \\ \rightsquigarrow \\ \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\!\vert\ \cup_{i \in L} \hat{\Upsilon}_i\end{array}\right)} \tag{5.5d}$$

$$\frac{\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\vert\ \hat{\Upsilon}}{\Delta \vdash_{\hat{\Phi}} \texttt{uinp}[\ell](\hat{p}) \rightsquigarrow \texttt{inp}[\ell](p) : \texttt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\!\vert\ \hat{\Upsilon}} \tag{5.5e}$$

The following rule governs upTSM application. It is written identically to Rule (4.6f).

$$\frac{b \downarrow e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{Success} \cdot e_{\text{cand}} \qquad e_{\text{cand}} \uparrow_{\mathsf{CEPat}} \grave{p} \qquad \vdash^{\Delta;\,\hat{\Phi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uptsm}(\tau;e_{\text{parse}});\,b}\ \grave{p} \rightsquigarrow p : \tau \dashv\!\!| ^{\hat{\Upsilon}}}{\Delta \vdash_{\hat{\Phi},\hat{a}\rightsquigarrow a\hookrightarrow \texttt{uptsm}(\tau;e_{\text{parse}})} \hat{a}\ /b/ \rightsquigarrow p : \tau \dashv\!\!| \hat{\Upsilon}} \tag{5.5f}$$

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 (5.5f).

$$\frac{b \downarrow e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{Success} \cdot e_{\text{cand}} \qquad e_{\text{cand}} \uparrow_{\mathsf{CEPat}} \grave{p} \qquad \vdash^{\Delta;\,\langle \mathcal{A};\Phi,a\hookrightarrow \texttt{uptsm}(\tau;e_{\text{parse}});\mathcal{I},\tau\hookrightarrow a\rangle;\,b}\ \grave{p} \rightsquigarrow p : \tau \dashv\!\!| ^{\hat{\Upsilon}}}{\Delta \vdash_{\langle \mathcal{A};\Phi,a\hookrightarrow \texttt{uptsm}(\tau;e_{\text{parse}});\mathcal{I},\tau\hookrightarrow a\rangle} /b/ \rightsquigarrow p : \tau \dashv\!\!| \hat{\Upsilon}} \tag{5.5g}$$

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 $\mathcal{U}(\Upsilon)$ was defined in 4.2.5.

Lemma 5.2 (Pattern Expressibility). *If $\Delta \vdash p : \tau \dashv\!\!| \Upsilon$ then $\Delta \vdash_{\hat{\Phi}} \mathcal{U}(p) \rightsquigarrow p : \tau \dashv\!\!| \mathcal{U}(\Upsilon)$.*

Proof. By rule induction over Rules (4.3), using the definitions of $\mathcal{U}(\Upsilon)$ and $\mathcal{U}(p)$. In each case, we can apply the IH to or over each premise, then apply the corresponding rule in Rules (5.5). □

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 5.3 (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 5.2 to each pattern typing premise, then derive the conclusion by applying Rules (5.1) and Rule (5.3). □

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

Figure 5.3: Abstract syntax of candidate expansion types, expressions, rules and patterns in miniVerse$_{\mathbf{U}}^{\mathbf{B}}$. Candidate expansion terms are identified up to α-equivalence.

5.2.4 Syntax of Candidate Expansions

Figure 5.3 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:

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

- 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*, $\texttt{cesplicedp}[m;n]$.

5.2.5 Bidirectional Candidate Expansion Validation

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

Judgement Form	Description
$\Delta \vdash^{\mathbb{T}} \grave{\tau} \leadsto \tau \;\mathsf{type}$	$\grave{\tau}$ is well-formed and has expansion τ assuming Δ and type splicing scene $\mathbb{T}$
$\Delta\; \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e \Rightarrow \tau$	$\grave{e}$ has expansion e and synthesizes type τ assuming Δ and Γ and expression splicing scene $\mathbb{E}$
$\Delta\; \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e \Leftarrow \tau$	$\grave{e}$ has expansion e when analyzed against type τ assuming Δ and Γ and expression splicing scene $\mathbb{E}$
$\Delta\; \Gamma \vdash^{\mathbb{E}} \grave{r} \leadsto r \Rightarrow \tau \Mapsto \tau'$	$\grave{r}$ has expansion r and takes values of type τ to values of synthesized type τ' assuming Δ and Γ and $\mathbb{E}$
$\Delta\; \Gamma \vdash^{\mathbb{E}} \grave{r} \leadsto r \Leftarrow \tau \Mapsto \tau'$	$\grave{r}$ 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 $\mathbb{E}$
$\vdash^{\mathbb{P}} \grave{p} \leadsto p : \tau \dashv\vert^{\hat{\Upsilon}}$	$\grave{p}$ expands to p and matches values of type τ generating assumptions $\hat{\Upsilon}$ 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*, $\Delta \vdash^{\mathbb{T}} \grave{\tau} \leadsto \tau \;\mathsf{type}$, 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*, $\Delta\; \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e \Rightarrow \tau$, and the *analytic ce-expression validation judgement*, $\Delta\; \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e \Leftarrow \tau$, are defined mutually inductively with Rules (5.1) and Rules (5.2) by Rules (5.6) and Rules (5.7), respectively, as follows.

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

$$\frac{}{\Delta\; \Gamma, x : \tau \vdash^{\mathbb{E}} x \leadsto x \Rightarrow \tau} \tag{5.6a}$$

$$\frac{\Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau} \leadsto \tau \ \mathsf{type} \qquad \Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e \Leftarrow \tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{ceasc}\{\grave{\tau}\}(\grave{e}) \leadsto e \Rightarrow \tau} \tag{5.6b}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e \Rightarrow \tau \qquad \Delta\ \Gamma, x : \tau \vdash^{\mathbb{E}} \grave{e}' \leadsto e' \Rightarrow \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{celetval}(\grave{e}; x.\grave{e}') \leadsto \mathtt{ap}(\mathtt{lam}\{\tau\}(x.e'); e) \Rightarrow \tau'} \tag{5.6c}$$

$$\frac{\Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau}_1 \leadsto \tau_1 \ \mathsf{type} \qquad \Delta\ \Gamma, x : \tau_1 \vdash^{\mathbb{E}} \grave{e} \leadsto e \Rightarrow \tau_2}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{celam}\{\grave{\tau}_1\}(x.\grave{e}) \leadsto \mathtt{lam}\{\tau_1\}(x.e) \Rightarrow \mathtt{parr}(\tau_1; \tau_2)} \tag{5.6d}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e}_1 \leadsto e_1 \Rightarrow \mathtt{parr}(\tau_2; \tau) \qquad \Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e}_2 \leadsto e_2 \Leftarrow \tau_2}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{ceap}(\grave{e}_1; \grave{e}_2) \leadsto \mathtt{ap}(e_1; e_2) \Rightarrow \tau} \tag{5.6e}$$

$$\frac{\Delta, t \ \mathsf{type}\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e \Rightarrow \tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{cetlam}(t.\grave{e}) \leadsto \mathtt{tlam}(t.e) \Rightarrow \mathtt{all}(t.\tau)} \tag{5.6f}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e \Rightarrow \mathtt{all}(t.\tau) \qquad \Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau}' \leadsto \tau' \ \mathsf{type}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{cetap}\{\grave{\tau}'\}(\grave{e}) \leadsto \mathtt{tap}\{\tau'\}(e) \Rightarrow [\tau'/t]\tau} \tag{5.6g}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e \Rightarrow \mathtt{rec}(t.\tau)}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{ceunfold}(\grave{e}) \leadsto \mathtt{unfold}(e) \Rightarrow [\mathtt{rec}(t.\tau)/t]\tau} \tag{5.6h}$$

$$\frac{\{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e}_i \leadsto e_i \Rightarrow \tau_i\}_{i \in L}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{cetpl}[L](\{i \hookrightarrow \grave{e}_i\}_{i \in L}) \leadsto \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) \Rightarrow \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{5.6i}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e \Rightarrow \mathtt{prod}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \mathtt{cepr}[\ell](\grave{e}) \leadsto \mathtt{pr}[\ell](e) \Rightarrow \tau} \tag{5.6j}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e \Rightarrow \tau \qquad \{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r}_i \leadsto r_i \Rightarrow \tau \Mapsto \tau'\}_{1 \le i \le n}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{cematch}[n](\grave{e}; \{\grave{r}_i\}_{1 \le i \le n}) \leadsto \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \le i \le n}) \Rightarrow \tau'} \tag{5.6k}$$

$$\frac{\begin{array}{c}\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e} \qquad \langle \mathcal{D}; \Delta_{\mathrm{app}} \rangle\ \langle \mathcal{G}; \Gamma_{\mathrm{app}} \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau \\ \Delta \cap \Delta_{\mathrm{app}} = \emptyset \qquad \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma_{\mathrm{app}}) = \emptyset\end{array}}{\Delta\ \Gamma \vdash^{\langle \mathcal{D}; \Delta_{\mathrm{app}} \rangle; \langle \mathcal{G}; \Gamma_{\mathrm{app}} \rangle; \hat{\Psi}; \hat{\Phi}; b} \mathtt{cesplicede}[m; n] \leadsto e \Rightarrow \tau} \tag{5.6l}$$

Rules (5.6a) through (5.6k) are analagous to Rules (5.1a) through (5.1k). Rule (5.6l) 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.

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e \Rightarrow \tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e \Leftarrow \tau} \tag{5.7a}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e \Rightarrow \tau \qquad \Delta\ \Gamma, x : \tau \vdash^{\mathbb{E}} \grave{e}' \leadsto e' \Leftarrow \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \mathtt{celetval}(\grave{e}; x.\grave{e}') \leadsto \mathtt{ap}(\mathtt{lam}\{\tau\}(x.e'); e) \Leftarrow \tau'} \tag{5.7b}$$

$$\frac{\Delta\ \Gamma, x : \tau_1 \vdash^{\mathbb{E}} \grave{e} \leadsto e \Leftarrow \tau_2}{\Delta\ \Gamma \vdash^{\mathbb{E}} \texttt{ceanalam}(x.\hat{e}) \leadsto \texttt{lam\{}\tau_1\texttt{\}}(x.e) \Leftarrow \texttt{parr}(\tau_1; \tau_2)} \tag{5.7c}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e \Leftarrow \tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \texttt{cetlam}(t.\grave{e}) \leadsto \texttt{tlam}(t.e) \Leftarrow \texttt{all}(t.\tau)} \tag{5.7d}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e \Leftarrow [\texttt{rec}(t.\tau)/t]\tau}{\Delta\ \Gamma \vdash^{\mathbb{E}} \texttt{cefold}(\grave{e}) \leadsto \texttt{fold\{}t.\tau\texttt{\}}(e) \Leftarrow \texttt{rec}(t.\tau)} \tag{5.7e}$$

$$\frac{\{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e}_i \leadsto e_i \Leftarrow \tau_i\}_{i \in L}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \texttt{cetpl}[L](\{i \hookrightarrow \grave{e}_i\}_{i \in L}) \leadsto \texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) \Leftarrow \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{5.7f}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e \Leftarrow \tau}{\left(\begin{array}{c}\Delta\ \Gamma \vdash^{\mathbb{E}} \texttt{cein}[\ell](\grave{e}) \\ \leadsto \\ \texttt{in}[L,\ell][\ell]\texttt{\{}\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau\texttt{\}}(e) \Leftarrow \texttt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)\end{array}\right)} \tag{5.7g}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e \Rightarrow \tau \qquad \{\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{r}_i \leadsto r_i \Leftarrow \tau \Mapsto \tau'\}_{1 \le i \le n}}{\Delta\ \Gamma \vdash^{\mathbb{E}} \texttt{cematch}[n](\grave{e}; \{\grave{r}_i\}_{1 \le i \le n}) \leadsto \texttt{match}[n]\texttt{\{}\tau'\texttt{\}}(e; \{r_i\}_{1 \le i \le n}) \Leftarrow \tau'} \tag{5.7h}$$

$$\frac{\begin{array}{c}\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e} \qquad \langle \mathcal{D}; \Delta_{\mathrm{app}} \rangle\ \langle \mathcal{G}; \Gamma_{\mathrm{app}} \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \leadsto e \Leftarrow \tau \\ \Delta \cap \Delta_{\mathrm{app}} = \emptyset \qquad \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma_{\mathrm{app}}) = \emptyset\end{array}}{\Delta\ \Gamma \vdash^{\langle \mathcal{D}; \Delta_{\mathrm{app}} \rangle; \langle \mathcal{G}; \Gamma_{\mathrm{app}} \rangle; \hat{\Psi}; \hat{\Phi}; b} \texttt{cesplicede}[m; n] \leadsto e \Leftarrow \tau} \tag{5.7i}$$

Rules (5.7a) through (5.7h) are analagous to Rules (5.2a) through (5.2h). Rule (5.7i) 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 (5.1) by the following rule.

$$\frac{\Delta \vdash p : \tau \dashv\!\mid \Upsilon \qquad \Delta\ \Gamma \cup \Upsilon \vdash^{\mathbb{E}} \grave{e} \leadsto e \Rightarrow \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \texttt{cerule}(p.\grave{e}) \leadsto \texttt{rule}(p.e) \Rightarrow \tau \Mapsto \tau'} \tag{5.8}$$

The *analytic ce-rule validation judgement* is defined mutually inductively with Rules (5.2) by the following rule.

$$\frac{\Delta \vdash p : \tau \dashv\!\mid \Upsilon \qquad \Delta\ \Gamma \cup \Upsilon \vdash^{\mathbb{E}} \grave{e} \leadsto e \Leftarrow \tau'}{\Delta\ \Gamma \vdash^{\mathbb{E}} \texttt{cerule}(p.\grave{e}) \leadsto \texttt{rule}(p.e) \Leftarrow \tau \Mapsto \tau'} \tag{5.9}$$

Candidate Expansion Pattern Validation

The *ce-pattern validation judgement* is inductively defined by the following rules, which are written identically to Rules (4.9).

$$\frac{}{\vdash^{\mathbb{P}} \texttt{cewildp} \leadsto \texttt{wildp} : \tau \dashv\!\mid^{\langle\emptyset;\emptyset\rangle}} \tag{5.10a}$$

$$\frac{\vdash^{\mathbb{P}} \grave{p} \leadsto p : [\texttt{rec}(t.\tau)/t]\tau \dashv\!\mid^{\hat{\Upsilon}}}{\vdash^{\mathbb{P}} \texttt{cefoldp}(\grave{p}) \leadsto \texttt{foldp}(p) : \texttt{rec}(t.\tau) \dashv\!\mid^{\hat{\Upsilon}}} \tag{5.10b}$$

$$\frac{\{\vdash^{\mathbb{P}} \grave{p}_i \leadsto p_i : \tau_i \dashv\!\mid^{\hat{\Upsilon}_i}\}_{i\in L}}{\left(\begin{array}{c}\vdash^{\mathbb{P}} \texttt{cetplp}[L](\{i \hookrightarrow \grave{p}_i\}_{i\in L}) \\ \leadsto \\ \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i\in L}) : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L}) \dashv\!\mid^{\cup_{i\in L}\hat{\Upsilon}_i}\end{array}\right)} \tag{5.10c}$$

$$\frac{\vdash^{\mathbb{P}} \grave{p} \leadsto p : \tau \dashv\!\mid^{\hat{\Upsilon}}}{\vdash^{\mathbb{P}} \texttt{ceinp}[\ell](\grave{p}) \leadsto \texttt{inp}[\ell](p) : \texttt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i\in L}; \ell \hookrightarrow \tau) \dashv\!\mid^{\hat{\Upsilon}}} \tag{5.10d}$$

$$\frac{\mathsf{parseUPat}(\mathsf{subseq}(b;m;n)) = \hat{p} \qquad \Delta \vdash_{\hat{\Phi}} \hat{p} \leadsto p : \tau \dashv\!\mid \hat{\Upsilon}}{\vdash^{\Delta;\hat{\Phi};b} \texttt{cesplicedp}[m;n] \leadsto p : \tau \dashv\!\mid^{\hat{\Upsilon}}} \tag{5.10e}$$

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

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

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 (5.6) and Rule (5.8) as needed. □

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

Lemma 5.5 (Candidate Expansion Pattern Expressibility)**.** *If $\Delta \vdash p : \tau \dashv\!\mid \emptyset$ then $\vdash^{\Delta;\hat{\Phi};b} \mathcal{C}(p) \leadsto p : \tau \dashv\!\mid^{\langle\emptyset;\emptyset\rangle}$.*

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 (5.10). □

5.2.6 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 5.6 (Typed Pattern Expansion)**.** *Both of the following hold:*

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

Proof. My mutual rule induction over Rules (5.5) and Rules (5.10).

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

   Case (5.5a) through (5.5f)**.** 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 (5.5g)**.** 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)	$e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{Success} \cdot e_{\text{cand}}$	by assumption
(6)	$e_{\text{cand}} \uparrow_{\text{CEPat}} \grave{p}$	by assumption
(7)	$\vdash^{\Delta; \langle \mathcal{A}; \Phi', a \hookrightarrow \texttt{uptsm}(\tau; e_{\text{parse}}); \mathcal{I}', \tau \hookrightarrow a \rangle; b} \grave{p} \rightsquigarrow p : \tau \dashv\!\mid^{\langle \mathcal{G}; \Upsilon \rangle}$	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 $\hat{\Phi} = \langle \mathcal{A}; \Phi; \mathcal{I} \rangle$.

   Case (5.10a) through (5.10e)**.** 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:

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

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 (5.5f) and (5.5g). There, we have that the metric remains stable:

$$\begin{aligned}
&\|\Delta \vdash_{\hat{\Phi}, \hat{a} \leadsto a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}})} \texttt{uapuptsm}[b][\hat{a}] \leadsto p : \tau \dashv\!| \hat{\Upsilon}\| \\
=&\|\Delta \vdash_{\langle \mathcal{A}; \Phi', a \hookrightarrow \texttt{uptsm}(\tau; e_{\text{parse}}); \mathcal{I}', \tau \hookrightarrow a\rangle} \texttt{uplit}[b] \leadsto p : \tau \dashv\!| \hat{\Upsilon}\| \\
=&\|\vdash^{\Delta; \hat{\Phi}, \hat{a} \leadsto a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}}); b} \grave{p} \leadsto p : \tau \dashv\!|^{\hat{\Upsilon}}\| \\
=&\|b\|
\end{aligned}$$

The only case in the proof of part 2 that invokes part 1 is Case (5.10e). There, we have that $\mathsf{parseUPat}(\mathsf{subseq}(b; m; n)) = \hat{p}$ and the IH is applied to the judgement $\Delta \vdash_{\hat{\Phi}} \hat{p} \leadsto p : \tau \dashv\!| \hat{\Upsilon}$. 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:

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

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 5.7 (Pattern Parsing Monotonicity)**.** *If* $\mathsf{parseUPat}(b) = \hat{p}$ *then* $\|\hat{p}\| < \|b\|$.

Combining Conditions 3.22 and 5.7, 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 5.8 (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* $\langle \mathcal{D}; \Delta\rangle\ \langle \mathcal{G}; \Gamma\rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau$ *then* $\Delta\ \Gamma \vdash e : \tau$.
   ii. If $\langle \mathcal{D}; \Delta\rangle\ \langle \mathcal{G}; \Gamma\rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{r} \leadsto r \Rightarrow \tau \Mapsto \tau'$ *then* $\Delta\ \Gamma \vdash r : \tau \Mapsto \tau'$.
   (b) *i. If* $\langle \mathcal{D}; \Delta\rangle\ \langle \mathcal{G}; \Gamma\rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \leadsto e \Leftarrow \tau$ *and* $\Delta \vdash \tau$ type *then* $\Delta\ \Gamma \vdash e : \tau$.
   ii. If $\langle \mathcal{D}; \Delta\rangle\ \langle \mathcal{G}; \Gamma\rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{r} \leadsto r \Leftarrow \tau \Mapsto \tau'$ *and* $\Delta \vdash \tau'$ type *then* $\Delta\ \Gamma \vdash r : \tau \Mapsto \tau'$.
2. *(a)* *i. If* $\Delta\ \Gamma \vdash^{\langle \mathcal{D}; \Delta_{app}\rangle; \langle \mathcal{G}; \Gamma_{app}\rangle; \hat{\Psi}; \hat{\Phi}; b} \grave{e} \leadsto e \Rightarrow \tau$ *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 $\Delta\ \Gamma \vdash^{\langle \mathcal{D}; \Delta_{app}\rangle; \langle \mathcal{G}; \Gamma_{app}\rangle; \hat{\Psi}; \hat{\Phi}; b} \grave{r} \leadsto r \Rightarrow \tau \Mapsto \tau'$ *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 (5.1), Rules (5.2), Rule (5.3), Rule (5.4), Rules (5.6), Rules (5.7), Rule (5.8) and Rule (5.9). 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 (5.1a)**.** 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 (5.1b)**.** We have:

(1) $\hat{e} = \texttt{uasc}\{\hat{\tau}\}(\hat{e}')$	by assumption
(2) $\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau$ type	by assumption
(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \rightsquigarrow e \Leftarrow \tau$	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 (5.1c) through (5.1k)**.** 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 (5.1l)**.** We have:

(1) $\hat{e} = \texttt{usyntaxue}\{e_{\text{parse}}\}\{\hat{\tau}'\}(\hat{a}.\hat{e}')$	by assumption
(2) $\hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'$ 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 (5.1m)**.** 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) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{Success} \cdot e_{\text{cand}}$ — by assumption

(5) $e_{\text{cand}} \uparrow_{\text{CEExp}} \grave{e}$ — 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) $\emptyset \cap \text{dom}(\Gamma) = \emptyset$ — by finite set intersection identity

(11) $\emptyset \cup \Delta\, \emptyset \cup \Gamma \vdash e : \tau$ — by IH, part 2(a)(i) on (6), (9), (10) and (8)

(12) $\Delta\, \Gamma \vdash e : \tau$ — by definition of finite set and finite function union over (11)

Case (5.1n)**.** 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 (5.1o)**.** 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 (5.1p)**.** 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) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a;\Phi, a \hookrightarrow \texttt{uptsm}(\tau';e_{\text{parse}});\mathcal{I} \uplus \tau \hookrightarrow a\rangle} \hat{e} \rightsquigarrow e \Rightarrow \tau$ 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 (5.3)**.** We have:

(1) $\hat{r} = \texttt{urule}(\hat{p}.\hat{e})$	by assumption
(2) $r = \texttt{rule}(p.e)$	by assumption
(3) $\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \langle \mathcal{A}';\Upsilon\rangle$	by assumption
(4) $\hat{\Delta}\ \langle \mathcal{A} \uplus \mathcal{A}';\Gamma \cup \Upsilon\rangle \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau'$	by assumption
(5) $\Delta \vdash p : \tau \dashv\!\mid \Upsilon$	by Theorem 5.6, part 1 on (3)
(6) $\Delta\ \Gamma \cup \Upsilon \vdash e : \tau'$	by IH, part 1(a)(i) on (4)
(7) $\Delta\ \Gamma \vdash \texttt{rule}(p.e) : \tau \Mapsto \tau'$	by Rule (4.2) on (5) and (6)

(b) i. We induct on the assumption.

Case (5.2a)**.** 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 (5.2b) through (5.2h)**.** 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 (5.2i)**.** We have:

(3) $\hat{e} = \texttt{usyntaxue\{}e_{\text{parse}}\texttt{\}\{}\hat{\tau}'\texttt{\}(}\hat{a}.\hat{e}'\texttt{)}$	by assumption
(4) $\hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'$ type	by assumption
(5) ,	by assumption
(6) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{uetsm}(\tau';e_{\text{parse}});\hat{\Phi}} \hat{e}' \rightsquigarrow e \Leftarrow \tau$	by assumption
(7) $\Delta \vdash \Psi$ ueTSMs	by assumption
(8) $\Delta \vdash \tau'$ type	by Lemma 3.12 to (4)
(9) $\Delta \vdash \Psi, \hat{a} \hookrightarrow \texttt{uetsm}(\tau'; e_{\text{parse}})$ 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 (5.2j). 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) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{Success} \cdot e_{\text{cand}}$ — by assumption

(5) $e_{\text{cand}} \uparrow_{\text{CEExp}} \grave{e}$ — 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) $\emptyset \cap \text{dom}(\Gamma) = \emptyset$ — 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 (5.2k). 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) $e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{Success} \cdot e_{\text{cand}}$ — by assumption

(5) $e_{\text{cand}} \uparrow_{\text{CEExp}} \grave{e}$ — 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) $\emptyset \cap \text{dom}(\Gamma) = \emptyset$ — 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 (5.2l). We have:

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

(2) $\hat{\Delta} \vdash \hat{\tau}' \rightsquigarrow \tau'$ 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 (5.2m)**.** We have:

(1) $\hat{e} = \texttt{uimplicitp}[\hat{a}](\hat{e})$	by assumption
(2) $\hat{\Phi} = \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{uptsm}(\tau'; e_{\text{parse}}); \mathcal{I}\rangle$	by assumption
(3) $\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{uptsm}(\tau'; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a\rangle} \hat{e} \rightsquigarrow e \Leftarrow \tau$	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 (5.4)**.** We have:

(1) $\hat{r} = \texttt{urule}(\hat{p}.\hat{e})$	by assumption
(2) $r = \texttt{rule}(p.e)$	by assumption
(3) $\Delta \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\vert \langle \mathcal{A}'; \Upsilon\rangle$	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) $\Delta \vdash p : \tau \dashv\vert \Upsilon$	by Theorem 5.6, part 1 on (3)
(6) $\Delta\ \Gamma \cup \Upsilon \vdash e : \tau'$	by IH, part 1(b)(i) on (4)
(7) $\Delta\ \Gamma \vdash \texttt{rule}(p.e) : \tau \Rightarrow \tau'$	by Rule (4.2) on (5) and (6)

2. In the following, let $\hat{\Delta} = \langle \mathcal{D}; \Delta_{\text{app}}\rangle$ and $\hat{\Gamma} = \langle \mathcal{G}; \Gamma_{\text{app}}\rangle$ and $\mathbb{E} = \hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; \hat{\Phi}; b$.

(a) i. We induct on the assumption.

Case (5.6a)**.** 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 (5.6b)**.** We have:

(1) $\grave{e} = \texttt{ceasc}\{\grave{\tau}\}(\grave{e}')$	by assumption
(2) $\Delta \cap \Delta_{\text{app}} = \emptyset$	by assumption
(3) $\text{dom}(\Gamma) \cap \text{dom}(\Gamma_{\text{app}}) = \emptyset$	by assumption
(4) $\Delta \vdash^{\mathsf{ts}(\mathbb{E})} \grave{\tau} \rightsquigarrow \tau\ \mathsf{type}$	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 (5.6c) through (5.6k)**.** 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 (5.6l). We have:

(1)	$\grave{e} = \texttt{cesplicede}[m; n]$	by assumption
(2)	$\mathsf{parseUExp}(\mathsf{subseq}(b; m; n)) = \hat{e}$	by assumption
(3)	$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau$	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 (5.8). 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)	$\Delta\ \Gamma \cup \Upsilon \vdash^{\hat{\Delta}; \hat{\Gamma}; \hat{\Psi}; \hat{\Phi}; b} \grave{e} \leadsto e \Rightarrow \tau'$	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)	$\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash \texttt{rule}(p.e) : \tau \Mapsto \tau'$	by Rule (4.2) on (3) and (11)

(b) i. We induct on the assumption.

Case (5.7a). We have:

(1)	$\Delta\ \Gamma \vdash^{\mathbb{E}} \grave{e} \leadsto e \Rightarrow \tau$	by assumption
(2)	$\Delta\ \Gamma \vdash e : \tau$	by IH, part 2(a)(i) on (1)

Case (5.7b) through (5.2h). 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 (5.7i). 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)	$\Delta \cup \Delta_{\text{app}} \vdash \tau\ \mathsf{type}$	by assumption
(7)	$\Delta \cap \Delta_{\text{app}} = \emptyset$	by assumption
(8)	$\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma_{\text{app}}) = \emptyset$	by assumption
(9)	$\Delta_{\text{app}}\ \Gamma_{\text{app}} \vdash e : \tau$	by IH, part 1(b)(i) on (5), (7), (8) and (6)
(10)	$\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash e : \tau$	by Lemma 4.1 over Δ and Γ and exchange on (9)

ii. We induct on the assumption. There is one case.

Case (5.9). We have:

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

(13) $\Delta \cup \Delta_{\text{app}}\ \Gamma \cup \Gamma_{\text{app}} \vdash \texttt{rule}(p.e) : \tau \Mapsto \tau'$ 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 (5.1m) in the proof of part 1(a)(i) and Case (5.2k) 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 (5.6l) in the proof of part 2(a)(i) and Case (5.7i) 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 $\hat{e}$,

$$\begin{aligned}
\|\hat{x}\| &= 0 \\
\|\texttt{uasc}\{\hat{\tau}\}(\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{uletval}(\hat{e};\hat{x}.\hat{e}')\| &= \|\hat{e}\| + \|\hat{e}'\| \\
\|\texttt{uanalam}(\hat{x}.\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{ulam}\{\hat{\tau}\}(\hat{x}.\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{uap}(\hat{e}_1;\hat{e}_2)\| &= \|\hat{e}_1\| + \|\hat{e}_2\| \\
\|\texttt{utlam}(\hat{t}.\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{utap}\{\hat{\tau}\}(\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{ufold}(\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{uunfold}(\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{utpl}[L](\{i \hookrightarrow \hat{e}_i\}_{i \in L})\| &= \sum_{i \in L} \|\hat{e}_i\| \\
\|\texttt{upr}[\ell](\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{uin}[\ell](\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{umatch}[n](\hat{e};\{\hat{r}_i\}_{1 \le i \le n})\| &= \|\hat{e}\| + \sum_{1 \le i \le n} \|r_i\| \\
\|\texttt{usyntaxue}\{e_{\text{parse}}\}\{\hat{\tau}\}(\hat{a}.\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{uimplicite}[\hat{a}](\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{uapuetsm}[b][\hat{a}]\| &= \|b\|
\end{aligned}$$

$$\|\texttt{uelit}[b]\| = \|b\|$$
$$\|\texttt{usyntaxup\{}e_{\text{parse}}\texttt{\}\{}\hat{\tau}\texttt{\}(}\hat{a}.\hat{e}\texttt{)}\| = \|\hat{e}\|$$
$$\|\texttt{uimplicitp[}\hat{a}\texttt{](}\hat{e}\texttt{)}\| = \|\hat{e}\|$$

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

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

Going from part 1 to part 2, the metric remains stable:

$$\begin{aligned}
&\|\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{uapuetsm}[b]\texttt{[}\hat{a}\texttt{]} \rightsquigarrow e \Rightarrow \tau\| \\
=&\|\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{uelit}[b] \rightsquigarrow e \Leftarrow \tau\| \\
=&\|\emptyset\ \emptyset \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi};\hat{\Phi};b} \grave{e} \rightsquigarrow e \Leftarrow \tau\| \\
=&\|b\|
\end{aligned}$$

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

Combining Conditions 3.22 and 5.9, we have that $\|\hat{e}\| < \|b\|$ as needed.

- In Case (5.2a) 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 (5.7a) 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.

□

5.3 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

Chapter 6

Parameterized TSMs (pTSMs)

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. 6.1 by introducing parameterized TSMs by example in VerseML. In particular, we discuss type parameters in Sec. 6.1.1 and module parameters in Sec. 6.1.2. We then formalize parameterized TSMs by defining a reduced calculus, $\mathsf{miniVerse}_{\forall}$, in Sec. 6.2.

6.1 Parameterized TSMs By Example

6.1.1 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*.

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`.

[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)`.

As discussed in Sec. 2.2.2, 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.

6.1.2 Module Parameters

VerseML also provides a module language based on the Standard ML module language [26]. 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 [17] 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:

```
val q : TwoListQueue.queue(int) =
  TwoListQueue.insert(1,
    TwoListQueue.insert(2,
      TwoListQueue.insert(3,
        TwoListQueue.empty)))
val q' : TwoListQueue.queue(int) =
  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
			$\texttt{unit}$	$\langle\!\langle\rangle\!\rangle$	nullary product
			$\texttt{dprod}(\kappa; u.\kappa)$	$\langle\!\langle u :: \kappa; \kappa\rangle\!\rangle$	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
			$\texttt{triv}$	$\langle\!\langle\rangle\!\rangle$	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 6.1: Syntax of kinds and constructors in $\mathsf{miniVerse}_\forall$. 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
			$\texttt{in}[L][\ell]\{\{i \hookrightarrow \tau_i\}_{i \in L}\}(e)$	$\ell \cdot e$	injection
			$\texttt{match}[n]\{\tau\}(e; \{r_i\}_{1 \leq i \leq n})$	$\texttt{match}\; e\; \{r_i\}_{1 \leq i \leq n}$	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
			$\texttt{inp}[\ell](p)$	$\ell \cdot p$	injection pattern

Figure 6.2: 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 6.3: Syntax of module expressions and signatures in miniVerse$_\forall$. Module expressions and signatures are identified up to α-equivalence.

6.2 $\text{miniVerse}_\forall$

6.2.1 Syntax of the Inner Language

Syntax of the Inner Core Language

We adopt the metatheoretic conventions established for our definitions of $\text{miniVerse}_{\mathbf{UE}}$ in Sec. 3.2 and $\text{miniVerse}_{\mathbf{U}}$ in Sec. 4.2 without restating them.

Kinds and constructors in Figure 6.1.

Expanded expressions, rules and patterns in Figure 6.2.

Syntax of the Inner Module Language

Module expressions and signatures in Figure 6.3.

6.2.2 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
$\Omega \vdash r : \tau \Mapsto \tau'$	r takes values of type τ to values of type τ'
$\Omega \vdash p : \tau \dashv\mid \Omega'$	p matches values of type τ and generates hypotheses Ω'

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
$\Omega \vdash \sigma \ \mathsf{sig}$	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
$\Omega \vdash M \ \mathsf{mval}$	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 $\Omega \vdash \kappa \ \mathsf{kind}$ for the extension of Ω with a mapping from u to $u :: \kappa$
- $\Omega, X : \sigma$ when $X \notin \mathrm{dom}(\Omega)$ and $\Omega \vdash \sigma \ \mathsf{sig}$ 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

$$\frac{\Omega \vdash \kappa_1 \ \mathsf{kind} \qquad \Omega, u :: \kappa_1 \vdash \kappa_2 \ \mathsf{kind}}{\Omega \vdash \mathtt{darr}(\kappa_1; u.\kappa_2) \ \mathsf{kind}} \tag{6.1a}$$

$$\frac{}{\Omega \vdash \mathtt{unit} \ \mathsf{kind}} \tag{6.1b}$$

$$\frac{\Omega \vdash \kappa_1 \ \mathsf{kind} \qquad \Omega, u :: \kappa_1 \vdash \kappa_2 \ \mathsf{kind}}{\Omega \vdash \mathtt{dprod}(\kappa_1; u.\kappa_2) \ \mathsf{kind}} \tag{6.1c}$$

$$\frac{}{\Omega \vdash \mathtt{Type} \ \mathsf{kind}} \tag{6.1d}$$

$$\frac{\Omega \vdash \tau :: \mathtt{Type}}{\Omega \vdash \mathtt{S}(\tau) \ \mathsf{kind}} \tag{6.1e}$$

Kind definitional equality TODO: describe these

$$\frac{\Omega \vdash \kappa \ \mathsf{kind}}{\Omega \vdash \kappa \equiv \kappa} \tag{6.2a}$$

$$\frac{\Omega \vdash \kappa \equiv \kappa'}{\Omega \vdash \kappa' \equiv \kappa} \tag{6.2b}$$

$$\frac{\Omega \vdash \kappa \equiv \kappa' \qquad \Omega \vdash \kappa' \equiv \kappa''}{\Omega \vdash \kappa \equiv \kappa''} \tag{6.2c}$$

$$\frac{\Omega \vdash \kappa_1 \equiv \kappa_1' \qquad \Omega, u :: \kappa_1 \vdash \kappa_2 \equiv \kappa_2'}{\Omega \vdash \mathtt{darr}(\kappa_1; u.\kappa_2) \equiv \mathtt{darr}(\kappa_1'; u.\kappa_2')} \tag{6.2d}$$

$$\frac{\Omega \vdash \kappa_1 \equiv \kappa_1' \qquad \Omega, u :: \kappa_1 \vdash \kappa_2 \equiv \kappa_2'}{\Omega \vdash \mathtt{dprod}(\kappa_1; u.\kappa_2) \equiv \mathtt{dprod}(\kappa_1'; u.\kappa_2')} \tag{6.2e}$$

$$\frac{\Omega \vdash c \equiv c' :: \mathtt{Type}}{\Omega \vdash \mathtt{S}(c) \equiv \mathtt{S}(c')} \tag{6.2f}$$

Subkinding TODO: describe these

$$\frac{\Omega \vdash \kappa \equiv \kappa'}{\Omega \vdash \kappa <:: \kappa'} \tag{6.3a}$$

$$\frac{\Omega \vdash \kappa <:: \kappa' \qquad \Omega \vdash \kappa' <:: \kappa''}{\Omega \vdash \kappa <:: \kappa''} \tag{6.3b}$$

$$\frac{\Omega \vdash \kappa_1' <:: \kappa_1 \qquad \Omega, u :: \kappa_1' \vdash \kappa_2 <:: \kappa_2'}{\Omega \vdash \mathtt{darr}(\kappa_1; u.\kappa_2) <:: \mathtt{darr}(\kappa_1'; u.\kappa_2')} \tag{6.3c}$$

$$\frac{\Omega \vdash \kappa_1 <:: \kappa_1' \qquad \Omega, u :: \kappa_1 \vdash \kappa_2 <:: \kappa_2'}{\Omega \vdash \mathtt{dprod}(\kappa_1; u.\kappa_2) <:: \mathtt{dprod}(\kappa_1'; u.\kappa_2')} \tag{6.3d}$$

$$\frac{\Omega \vdash \tau :: \mathtt{Type}}{\Omega \vdash \mathtt{S}(\tau) <:: \mathtt{Type}} \tag{6.3e}$$

$$\frac{\Omega \vdash \tau <: \tau'}{\Omega \vdash \mathtt{S}(\tau) <:: \mathtt{S}(\tau')\mathtt{Type}} \tag{6.3f}$$

Kinding TODO: describe these

$$\frac{\Omega \vdash c :: \kappa_1 \qquad \Omega \vdash \kappa_1 <:: \kappa_2}{\Omega \vdash c :: \kappa_2} \tag{6.4a}$$

$$\frac{}{\Omega, u :: \kappa \vdash u :: \kappa} \tag{6.4b}$$

$$\frac{\Omega, u :: \kappa_1 \vdash c_2 :: \kappa_2}{\Omega \vdash \mathtt{abs}(u.c_2) :: \mathtt{darr}(\kappa_1; u.\kappa_2)} \tag{6.4c}$$

$$\frac{\Omega \vdash c_1 :: \mathtt{darr}(\kappa_2; u.\kappa) \qquad \Omega \vdash c_2 :: \kappa_2}{\Omega \vdash \mathtt{app}(c_1; c_2) :: [c_1/u]\kappa} \tag{6.4d}$$

$$\frac{}{\Omega \vdash \mathtt{triv} :: \mathtt{unit}} \tag{6.4e}$$

$$\frac{\Omega \vdash c_1 :: \kappa_1 \qquad \Omega \vdash c_2 :: [c_1/u]\kappa_2}{\Omega \vdash \mathtt{pair}(c_1; c_2) :: \mathtt{dprod}(\kappa_1; u.\kappa_2)} \tag{6.4f}$$

$$\frac{\Omega \vdash c :: \mathtt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash \mathtt{prl}(c) :: \kappa_1} \tag{6.4g}$$

$$\frac{\Omega \vdash c :: \mathtt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash \mathtt{prr}(c) :: [\mathtt{prl}(c)/u]\kappa_2} \tag{6.4h}$$

$$\frac{\Omega \vdash \tau_1 :: \mathsf{Type} \qquad \Omega \vdash \tau_2 :: \mathsf{Type}}{\Omega \vdash \mathtt{parr}(\tau_1; \tau_2) :: \mathsf{Type}} \tag{6.4i}$$

$$\frac{\Omega \vdash \kappa\ \mathsf{kind} \qquad \Omega, u :: \kappa \vdash \tau :: \mathsf{Type}}{\Omega \vdash \mathtt{all}\{\kappa\}(u.\tau) :: \mathsf{Type}} \tag{6.4j}$$

$$\frac{\Omega, t :: \mathsf{Type} \vdash \tau :: \mathsf{Type}}{\Omega \vdash \mathtt{rec}(t.\tau) :: \mathsf{Type}} \tag{6.4k}$$

$$\frac{\{\Omega \vdash \tau_i :: \mathsf{Type}\}_{1 \le i \le n}}{\Omega \vdash \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) :: \mathsf{Type}} \tag{6.4l}$$

$$\frac{\{\Omega \vdash \tau_i :: \mathsf{Type}\}_{1 \le i \le n}}{\Omega \vdash \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) :: \mathsf{Type}} \tag{6.4m}$$

$$\frac{\Omega \vdash c :: \mathsf{Type}}{\Omega \vdash c :: \mathsf{S}(c)} \tag{6.4n}$$

$$\frac{\Omega \vdash M\ \mathsf{mval} \qquad \Omega \vdash M : \mathtt{sig}\{\kappa\}(u.\tau)}{\Omega \vdash \mathtt{con}(M) :: \kappa} \tag{6.4o}$$

Constructor equalityTODO: describe this

$$\frac{\Omega \vdash c :: \kappa}{\Omega \vdash c \equiv c :: \kappa} \tag{6.5a}$$

$$\frac{\Omega \vdash c \equiv c' :: \kappa}{\Omega \vdash c' \equiv c :: \kappa} \tag{6.5b}$$

$$\frac{\Omega \vdash c \equiv c' :: \kappa \qquad \Omega \vdash c' \equiv c'' :: \kappa}{\Omega \vdash c \equiv c'' :: \kappa} \tag{6.5c}$$

$$\frac{\Omega, u :: \kappa_1 \vdash c \equiv c' :: \kappa_2}{\Omega \vdash \mathtt{abs}(u.c) \equiv \mathtt{abs}(u.c') :: \mathtt{darr}(\kappa_1; u.\kappa_2)} \tag{6.5d}$$

$$\frac{\Omega \vdash c_1 \equiv c_1' :: \mathtt{darr}(\kappa_2; u.\kappa) \qquad \Omega \vdash c_2 \equiv c_2' :: \kappa_2}{\Omega \vdash \mathtt{app}(c_1; c_2) \equiv \mathtt{app}(c_1'; c_2') :: \kappa} \tag{6.5e}$$

$$\frac{\Omega \vdash \texttt{abs}(u.c) :: \texttt{darr}(\kappa_2; u.\kappa) \qquad \Omega \vdash c_2 :: \kappa_2}{\Omega \vdash \texttt{app}(\texttt{abs}(u.c); c_2) \equiv [c_2/u]c :: [c_2/u]\kappa} \tag{6.5f}$$

$$\frac{\Omega \vdash c_1 \equiv c_1' :: \kappa_1 \qquad \Omega \vdash c_2 \equiv c_2' :: [c_1/u]\kappa_2}{\Omega \vdash \texttt{pair}(c_1; c_2) \equiv \texttt{pair}(c_1'; c_2') :: \texttt{dprod}(\kappa_1; u.\kappa_2)} \tag{6.5g}$$

$$\frac{\Omega \vdash c \equiv c' :: \texttt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash \texttt{prl}(c) \equiv \texttt{prl}(c') :: \kappa_1} \tag{6.5h}$$

$$\frac{\Omega \vdash c_1 :: \kappa_1 \qquad \Omega \vdash c_2 :: \kappa_2}{\Omega \vdash \texttt{prl}(\texttt{pair}(c_1; c_2)) \equiv c_1 :: \kappa_1} \tag{6.5i}$$

$$\frac{\Omega \vdash c \equiv c' :: \texttt{dprod}(\kappa_1; u.\kappa_2)}{\Omega \vdash \texttt{prr}(c) \equiv \texttt{prr}(c') :: [\texttt{prl}(c)/u]\kappa_2} \tag{6.5j}$$

$$\frac{\Omega \vdash c_1 :: \kappa_1 \qquad \Omega \vdash c_2 :: \kappa_2}{\Omega \vdash \texttt{prr}(\texttt{pair}(c_1; c_2)) \equiv c_2 :: \kappa_2} \tag{6.5k}$$

$$\frac{\Omega \vdash \tau_1 \equiv \tau_1' :: \texttt{Type} \qquad \Omega \vdash \tau_2 \equiv \tau_2' :: \texttt{Type}}{\Omega \vdash \texttt{parr}(\tau_1; \tau_2) \equiv \texttt{parr}(\tau_1'; \tau_2') :: \texttt{Type}} \tag{6.5l}$$

$$\frac{\Omega \vdash \kappa \equiv \kappa' \qquad \Omega, u :: \kappa \vdash \tau \equiv \tau' :: \texttt{Type}}{\Omega \vdash \texttt{all}\{\kappa\}(u.\tau) \equiv \texttt{all}\{\kappa'\}(u.\tau') :: \texttt{Type}} \tag{6.5m}$$

$$\frac{\Omega, t :: \texttt{Type} \vdash \tau \equiv \tau' :: \texttt{Type}}{\Omega \vdash \texttt{rec}(t.\tau) \equiv \texttt{rec}(t.\tau') :: \texttt{Type}} \tag{6.5n}$$

$$\frac{\{\Omega \vdash \tau_i \equiv \tau_i' :: \texttt{Type}\}_{1 \le i \le n}}{\Omega \vdash \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \equiv \texttt{prod}[L](\{i \hookrightarrow \tau_i'\}_{i \in L}) :: \texttt{Type}} \tag{6.5o}$$

$$\frac{\{\Omega \vdash \tau_i \equiv \tau_i' :: \texttt{Type}\}_{1 \le i \le n}}{\Omega \vdash \texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \equiv \texttt{sum}[L](\{i \hookrightarrow \tau_i'\}_{i \in L}) :: \texttt{Type}} \tag{6.5p}$$

$$\frac{\Omega \vdash c :: \texttt{S}(c')}{\Omega \vdash c \equiv c' :: \texttt{Type}} \tag{6.5q}$$

$$\frac{\Omega \vdash \texttt{struct}(c; e)\ \mathsf{mval} \qquad \Omega \vdash \texttt{struct}(c; e) : \texttt{sig}\{\kappa\}(u.\tau)}{\Omega \vdash \texttt{con}(\texttt{struct}(c; e)) \equiv c :: \kappa} \tag{6.5r}$$

Expanded Expressions, Rules and Patterns

Types, τ, classify expressions. The constructors of kind Type coincide with the types of miniVerse$_\forall$.

$$\frac{\Omega \vdash \tau :: \texttt{Type}}{\Omega \vdash \tau\ \mathsf{type}} \tag{6.6}$$

Type equality then coincides with constructor equality at kind Type.

$$\frac{\Omega \vdash \tau \equiv \tau :: \mathsf{Type}}{\Omega \vdash \tau \equiv \tau' \ \mathsf{type}} \tag{6.7}$$

Subtyping.

$$\frac{\Omega \vdash \tau_1 \equiv \tau_2 \ \mathsf{type}}{\Omega \vdash \tau_1 <: \tau_2} \tag{6.8a}$$

$$\frac{\Omega \vdash \tau <: \tau' \qquad \Omega \vdash \tau' <: \tau''}{\Omega \vdash \tau <: \tau''} \tag{6.8b}$$

$$\frac{\Omega \vdash \tau_1' <: \tau_1 \qquad \Omega \vdash \tau_2 <: \tau_2'}{\Omega \vdash \mathtt{parr}(\tau_1;\tau_2) <: \mathtt{parr}(\tau_1';\tau_2')} \tag{6.8c}$$

$$\frac{\Omega \vdash \kappa <:: \kappa' \qquad \Omega, u :: \kappa \vdash \tau <: \tau'}{\Omega \vdash \mathtt{all}\{\kappa\}(u.\tau) <: \mathtt{all}\{\kappa'\}(u.\tau')} \tag{6.8d}$$

$$\frac{\{\Omega \vdash \tau_i <: \tau_i'\}_{i \in L}}{\Omega \vdash \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) <: \mathtt{prod}[L](\{i \hookrightarrow \tau_i'\}_{i \in L})} \tag{6.8e}$$

$$\frac{\{\Omega \vdash \tau_i <: \tau_i'\}_{i \in L}}{\Omega \vdash \mathtt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) <: \mathtt{sum}[L](\{i \hookrightarrow \tau_i'\}_{i \in L})} \tag{6.8f}$$

Expression typing TODO: describe these

$$\frac{\Omega \vdash e : \tau \qquad \Omega \vdash \tau <: \tau'}{\Omega \vdash e : \tau'} \tag{6.9a}$$

$$\frac{}{\Omega, x : \tau \vdash x : \tau} \tag{6.9b}$$

$$\frac{\Omega \vdash \tau \ \mathsf{type} \qquad \Omega, x : \tau \vdash e : \tau'}{\Omega \vdash \mathtt{lam}\{\tau\}(x.e) : \mathtt{parr}(\tau;\tau')} \tag{6.9c}$$

$$\frac{\Omega \vdash e_1 : \mathtt{parr}(\tau;\tau') \qquad \Omega \vdash e_2 : \tau}{\Omega \vdash \mathtt{ap}(e_1;e_2) : \tau'} \tag{6.9d}$$

$$\frac{\Omega \vdash \kappa \ \mathsf{kind} \qquad \Omega, u :: \kappa \vdash e : \tau}{\Omega \vdash \mathtt{clam}\{\kappa\}(u.e) : \mathtt{all}\{\kappa\}(u.\tau)} \tag{6.9e}$$

$$\frac{\Omega \vdash e : \mathtt{all}\{\kappa\}(u.\tau) \qquad \Omega \vdash c :: \kappa}{\Omega \vdash \mathtt{cap}\{c\}(e) : [c/u]\tau} \tag{6.9f}$$

$$\frac{\Omega, t :: \mathsf{Type} \vdash \tau \ \mathsf{type} \qquad \Omega \vdash e : [\mathtt{rec}(t.\tau)/t]\tau}{\Omega \vdash \mathtt{fold}\{t.\tau\}(e) : \mathtt{rec}(t.\tau)} \tag{6.9g}$$

$$\frac{\Omega \vdash e : \mathtt{rec}(t.\tau)}{\Omega \vdash \mathtt{unfold}(e) : [\mathtt{rec}(t.\tau)/t]\tau} \tag{6.9h}$$

$$\frac{\{\Omega \vdash e_i : \tau_i\}_{i \in L}}{\Omega \vdash \mathtt{tpl}[L](\{i \hookrightarrow e_i\}_{i \in L}) : \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{6.9i}$$

$$\frac{\Omega \vdash e : \mathtt{prod}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)}{\Omega \vdash \mathtt{pr}[\ell](e) : \tau} \tag{6.9j}$$

$$\frac{\{\Omega \vdash \tau_i \ \mathsf{type}\}_{i \in L} \qquad \Omega \vdash \tau \ \mathsf{type} \qquad \Omega \vdash e : \tau}{\Omega \vdash \mathtt{in}[L, \ell][\ell]\{\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau\}(e) : \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau)} \tag{6.9k}$$

$$\frac{\Omega \vdash e : \tau \qquad \Omega \vdash \tau' \ \mathsf{type} \qquad \{\Omega \vdash r_i : \tau \Mapsto \tau'\}_{1 \le i \le n}}{\Omega \vdash \mathtt{match}[n]\{\tau'\}(e; \{r_i\}_{1 \le i \le n}) : \tau'} \tag{6.9l}$$

$$\frac{\Omega \vdash M \ \mathsf{mval} \qquad \Omega \vdash M : \mathtt{sig}\{\kappa\}(u.\tau)}{\Omega \vdash \mathtt{val}(M) : [\mathtt{con}(M)/u]\tau} \tag{6.9m}$$

Rule typing

$$\frac{\Omega \vdash p : \tau \dashv\!\!\mid \Omega' \qquad \Omega \cup \Omega' \vdash e : \tau'}{\Omega \vdash \mathtt{rule}(p.e) : \tau \Mapsto \tau'} \tag{6.10}$$

Pattern typing

$$\frac{}{\Omega \vdash x : \tau \dashv\!\!\mid x : \tau} \tag{6.11a}$$

$$\frac{}{\Omega \vdash \mathtt{wildp} : \tau \dashv\!\!\mid \emptyset} \tag{6.11b}$$

$$\frac{\Omega \vdash p : [\mathtt{rec}(t.\tau)/t]\tau \dashv\!\!\mid \Omega'}{\Omega \vdash \mathtt{foldp}(p) : \mathtt{rec}(t.\tau) \dashv\!\!\mid \Omega'} \tag{6.11c}$$

$$\frac{\{\Omega \vdash p_i : \tau_i \dashv\!\!\mid \Omega_i\}_{i \in L}}{\Omega \vdash \mathtt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) : \mathtt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv\!\!\mid \cup_{i \in L} \Omega_i} \tag{6.11d}$$

$$\frac{\Omega \vdash p : \tau \dashv\!\!\mid \Omega'}{\Omega \vdash \mathtt{inp}[\ell](p) : \mathtt{sum}[L, \ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\!\!\mid \Omega'} \tag{6.11e}$$

Signatures and Structures

Signature formation

$$\frac{\Omega \vdash \kappa \ \mathsf{kind} \qquad \Omega, u :: \kappa \vdash \tau \ \mathsf{type}}{\Omega \vdash \mathtt{sig}\{\kappa\}(u.\tau) \ \mathsf{sig}} \tag{6.12}$$

Signature equality

$$\frac{\Omega \vdash \kappa \equiv \kappa' \qquad \Omega, u :: \kappa \vdash \tau \equiv \tau' \ \mathsf{type}}{\Omega \vdash \mathtt{sig}\{\kappa\}(u.\tau) \equiv \mathtt{sig}\{\kappa'\}(u.\tau')} \tag{6.13}$$

Subsignature

$$\frac{\Omega \vdash \kappa <:: \kappa' \qquad \Omega, u :: \kappa \vdash \tau <: \tau'}{\Omega \vdash \mathtt{sig}\{\kappa\}(u.\tau) <: \mathtt{sig}\{\kappa'\}(u.\tau')} \tag{6.14}$$

Signature matching

$$\frac{\Omega \vdash M : \sigma \qquad \Omega \vdash \sigma <: \sigma'}{\Omega \vdash M : \sigma'} \tag{6.15a}$$

$$\frac{}{\Omega, X : \sigma \vdash X : \sigma} \tag{6.15b}$$

$$\frac{\Omega \vdash c :: \kappa \qquad \Omega \vdash e : [c/u]\tau}{\Omega \vdash \mathtt{struct}(c; e) : \mathtt{sig}\{\kappa\}(u.\tau)} \tag{6.15c}$$

$$\frac{\Omega \vdash \sigma \ \mathsf{sig} \qquad \Omega \vdash M : \sigma}{\Omega \vdash \mathtt{seal}\{\sigma\}(M) : \sigma} \tag{6.15d}$$

$$\frac{\Omega \vdash M : \sigma \qquad \Omega \vdash \sigma' \ \mathsf{sig} \qquad \Omega, X : \sigma \vdash M' : \sigma'}{\Omega \vdash \mathtt{mlet}\{\sigma'\}(M; X.M') : \sigma'} \tag{6.15e}$$

Module values

$$\frac{}{\Omega \vdash \mathtt{struct}(c; e) \ \mathsf{mval}} \tag{6.16a}$$

$$\frac{}{\Omega, X : \sigma \vdash X \ \mathsf{mval}} \tag{6.16b}$$

TODO: metatheory of statics of inner core

6.2.3 Structural Dynamics

TODO: do this

6.2.4 Syntax of the Surface Language

Syntax of the Surface Core Language

Unexpanded kinds and constructors in Figure 6.4
Unexpanded expressions, rules and patterns in Figure 6.5

Syntax of the TSM Language

Macro types, unexpanded macro types, macro expressions and unexpanded macro expressions in Figure 6.7

Syntax of Type Patterns

TODO: do this

Syntax of the Surface Module Language

Figure 6.6

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

Figure 6.4: Syntax of unexpanded kinds and constructors in miniVerse$_\forall$. By convention, we choose the metavariable $\hat{\tau}$ for constructors that, in well-formed terms, must necessarily expand to constructors of kind 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 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
			$\mathtt{uasc}\{\hat{\tau}\}(\hat{e})$	$\hat{e} : \hat{\tau}$	ascription
			$\mathtt{uletval}(\hat{e}; \hat{x}.\hat{e})$	$\mathtt{let}\ \mathtt{val}\ \hat{x} = \hat{e}\ \mathtt{in}\ \hat{e}$	value binding
			$\mathtt{uanalam}(\hat{x}.\hat{e})$	$\lambda\hat{x}.\hat{e}$	abstraction (unannotated)
			$\mathtt{ulam}\{\hat{\tau}\}(\hat{x}.\hat{e})$	$\lambda\hat{x}{:}\hat{\tau}.\hat{e}$	abstraction (annotated)
			$\mathtt{uap}(\hat{e}; \hat{e})$	$\hat{e}(\hat{e})$	application
			$\mathtt{uclam}\{\hat{\kappa}\}(\hat{u}.\hat{e})$	$\Lambda\hat{u}{::}\hat{\kappa}.\hat{e}$	constructor abstraction
			$\mathtt{ucap}\{\hat{c}\}(\hat{e})$	$\hat{e}[\hat{c}]$	constructor application
			$\mathtt{ufold}(\hat{e})$	$\mathtt{fold}(\hat{e})$	fold
			$\mathtt{uunfold}(\hat{e})$	$\mathtt{unfold}(\hat{e})$	unfold
			$\mathtt{utpl}[L](\{i \hookrightarrow \hat{e}_i\}_{i \in L})$	$\langle\{i \hookrightarrow \hat{e}_i\}_{i \in L}\rangle$	labeled tuple
			$\mathtt{upr}[\ell](\hat{e})$	$\hat{e} \cdot \ell$	projection
			$\mathtt{uin}[\ell](\hat{e})$	$\ell \cdot \hat{e}$	injection
			$\mathtt{umatch}[n](\hat{e}; \{\hat{r}_i\}_{1 \leq i \leq n})$	$\mathtt{match}\ \hat{e}\ \{\hat{r}_i\}_{1 \leq i \leq n}$	match
			$\mathtt{uval}(\hat{X})$	$\hat{X} \cdot \mathtt{v}$	value part
			$\mathtt{usyntaxpe}\{e\}\{\hat{\rho}\}(\hat{a}.\hat{e})$	$\mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for}$ $\mathtt{expressions}\ \{e\}\ \mathtt{in}\ \hat{e}$	peTSM definition
			$\mathtt{uletpetsm}\{\hat{e}\}(\hat{a}.\hat{e})$	$\mathtt{let}\ \mathtt{syntax}\ \hat{a} = \hat{e}\ \mathtt{for}$ $\mathtt{expressions}\ \mathtt{in}\ \hat{e}$	peTSM binding
			...	...	peTSM designation
			$\mathtt{uappetsm}[b]\{\hat{e}\}$	$\hat{e}\ /b/$	peTSM application
			$\mathtt{uelit}[b]$	$/b/$	peTSM unadorned literal
			$\mathtt{usyntaxpp}\{e\}\{\hat{\rho}\}(\hat{a}.\hat{e})$	$\mathtt{syntax}\ \hat{a}\ \mathtt{at}\ \hat{\rho}\ \mathtt{for}$ $\mathtt{patterns}\ \{e\}\ \mathtt{in}\ \hat{e}$	ppTSM definition
			$\mathtt{uletpptsm}\{\hat{e}\}(\hat{a}.\hat{e})$	$\mathtt{let}\ \mathtt{syntax}\ \hat{a} = \hat{e}\ \mathtt{for}$ $\mathtt{patterns}\ \mathtt{in}\ \hat{e}$	ppTSM binding
			...	...	ppTSM designation
URule	$\hat{r}$	$::=$	$\mathtt{urule}(\hat{p}.\hat{e})$	$\hat{p} \Rightarrow \hat{e}$	match rule
UPat	$\hat{p}$	$::=$	$\hat{x}$	$\hat{x}$	sigil pattern
			$\mathtt{uwildp}$	_	wildcard pattern
			$\mathtt{ufoldp}(\hat{p})$	$\mathtt{fold}(\hat{p})$	fold pattern
			$\mathtt{utplp}[L](\{i \hookrightarrow \hat{p}_i\}_{i \in L})$	$\langle\{i \hookrightarrow \hat{p}_i\}_{i \in L}\rangle$	labeled tuple pattern
			$\mathtt{uinp}[\ell](\hat{p})$	$\ell \cdot \hat{p}$	injection pattern
			$\mathtt{uappptsm}[b]\{\hat{e}\}$	$\hat{e}\ /b/$	ppTSM application
			$\mathtt{uplit}[b]$	$/b/$	ppTSM unadorned literal

Figure 6.5: Abstract syntax of unexpanded expressions, rules and patterns in miniVerse$_\forall$.

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

Figure 6.6: Abstract syntax of unexpanded module expressions and signatures in miniVerse$_\forall$.

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

Figure 6.7: Abstract syntax of TSM types, unexpanded TSM types, TSM expressions and unexpanded TSM expressions in miniVerse$_\forall$.

6.2.5 Typed Expansion

Unexpanded kinds and constructors

Judgement Form	**Description**
$\hat{\Omega} \vdash \hat{\kappa} \rightsquigarrow \kappa\ \mathsf{kind}$	$\hat{\kappa}$ is well-formed and has expansion κ
$\hat{\Omega} \vdash \hat{c} \rightsquigarrow c :: \kappa$	$\hat{c}$ has expansion c and kind κ

Unexpanded types, expressions, rules and patterns

Judgement Form	**Description**
$\hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type}$	$\hat{\tau}$ 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
$\Omega \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv\!\mid \hat{\Omega}$	$\hat{p}$ has expansion p and type τ and generates hypotheses $\hat{\Omega}$

Unexpanded signatures and module expressions

Judgement Form	**Description**
$\hat{\Omega} \vdash \hat{\sigma} \rightsquigarrow \sigma\ \mathsf{sig}$	$\hat{\sigma}$ has expansion σ
$\hat{\Omega} \vdash \hat{M} \rightsquigarrow M : \sigma$	$\hat{M}$ has expansion M and matches 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 $\hat{u} \in \text{dom}(\mathcal{D})$ 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 $\hat{x} \in \text{dom}(\mathcal{G})$ 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 $\hat{X} \in \text{dom}(\mathcal{M})$ 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*, $a \hookrightarrow \mathtt{petsm}(\rho; e_{\text{parse}})$ 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*, $a \hookrightarrow \mathtt{pptsm}(\rho; e_{\text{parse}})$ for some TSM type ρ and expanded expression e_{parse}.

Kinds and Constructors

Kind expansion

$$\frac{\hat{\Omega} \vdash \hat{\kappa}_1 \rightsquigarrow \kappa_1 \ \mathsf{kind} \qquad \hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa_1 \vdash \hat{\kappa}_2 \rightsquigarrow \kappa_2 \ \mathsf{kind}}{\hat{\Omega} \vdash \mathtt{udarr}(\hat{\kappa}_1; \hat{u}.\hat{\kappa}_2) \rightsquigarrow \mathtt{darr}(\kappa_1; u.\kappa_2) \ \mathsf{kind}} \tag{6.17a}$$

$$\frac{}{\hat{\Omega} \vdash \mathtt{uunit} \rightsquigarrow \mathtt{unit} \ \mathsf{kind}} \tag{6.17b}$$

$$\frac{\hat{\Omega} \vdash \hat{\kappa}_1 \rightsquigarrow \kappa_1 \ \mathsf{kind} \qquad \hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa_1 \vdash \hat{\kappa}_2 \rightsquigarrow \kappa_2 \ \mathsf{kind}}{\hat{\Omega} \vdash \mathtt{udprod}(\hat{\kappa}_1; \hat{u}.\hat{\kappa}_2) \rightsquigarrow \mathtt{dprod}(\kappa_1; u.\kappa_2) \ \mathsf{kind}} \tag{6.17c}$$

$$\frac{}{\hat{\Omega} \vdash \mathtt{uType} \rightsquigarrow \mathtt{Type} \ \mathsf{kind}} \tag{6.17d}$$

$$\frac{\hat{\Omega} \vdash \hat{\tau} \rightsquigarrow \tau :: \mathtt{Type}}{\hat{\Omega} \vdash \mathtt{uS}(\hat{\tau}) \rightsquigarrow \mathtt{S}(\tau) \ \mathsf{kind}} \tag{6.17e}$$

Constructor expansion

$$\frac{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c :: \kappa_1 \qquad \Omega \vdash \kappa_1 <:: \kappa_2}{\hat{\Omega} \vdash \hat{c} \rightsquigarrow c :: \kappa_2} \tag{6.18a}$$

$$\frac{}{\hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa \vdash \hat{u} \rightsquigarrow u :: \kappa} \tag{6.18b}$$

$$\frac{\hat{\Omega}, \hat{u} \rightsquigarrow u :: \kappa_1 \vdash \hat{c}_2 \rightsquigarrow c_2 :: \kappa_2}{\hat{\Omega} \vdash \mathtt{uabs}(\hat{u}.\hat{c}_2) \rightsquigarrow \mathtt{abs}(u.c_2) :: \mathtt{darr}(\kappa_1; u.\kappa_2)} \tag{6.18c}$$

$$\frac{\hat{\Omega} \vdash \hat{c}_1 \rightsquigarrow c_1 :: \mathtt{darr}(\kappa_2; u.\kappa) \qquad \hat{\Omega} \vdash \hat{c}_2 \rightsquigarrow c_2 :: \kappa_2}{\hat{\Omega} \vdash \mathtt{uapp}(\hat{c}_1; \hat{c}_2) \rightsquigarrow \mathtt{app}(c_1; c_2) :: [c_1/u]\kappa} \tag{6.18d}$$

$$\frac{}{\hat{\Omega} \vdash \mathtt{utriv} \rightsquigarrow \mathtt{triv} :: \mathtt{unit}} \tag{6.18e}$$

$$\frac{\hat{\Omega} \vdash \hat{c}_1 \leadsto c_1 :: \kappa_1 \qquad \hat{\Omega} \vdash \hat{c}_2 \leadsto c_2 :: [c_1/u]\kappa_2}{\hat{\Omega} \vdash \texttt{upair}(\hat{c}_1; \hat{c}_2) \leadsto \texttt{pair}(c_1; c_2) :: \texttt{dprod}(\kappa_1; u.\kappa_2)} \tag{6.18f}$$

$$\frac{\hat{\Omega} \vdash \hat{c} \leadsto c :: \texttt{dprod}(\kappa_1; u.\kappa_2)}{\hat{\Omega} \vdash \texttt{uprl}(\hat{c}) \leadsto \texttt{prl}(c) :: \kappa_1} \tag{6.18g}$$

$$\frac{\hat{\Omega} \vdash \hat{c} \leadsto c :: \texttt{dprod}(\kappa_1; u.\kappa_2)}{\hat{\Omega} \vdash \texttt{uprr}(\hat{c}) \leadsto \texttt{prr}(c) :: [\texttt{prl}(c)/u]\kappa_2} \tag{6.18h}$$

$$\frac{\hat{\Omega} \vdash \hat{\tau}_1 \leadsto \tau_1 :: \mathsf{Type} \qquad \hat{\Omega} \vdash \hat{\tau}_2 \leadsto \tau_2 :: \mathsf{Type}}{\hat{\Omega} \vdash \texttt{uparr}(\hat{\tau}_1; \hat{\tau}_2) \leadsto \texttt{parr}(\tau_1; \tau_2) :: \mathsf{Type}} \tag{6.18i}$$

$$\frac{\hat{\Omega} \vdash \hat{\kappa} \leadsto \kappa \ \mathsf{kind} \qquad \hat{\Omega}, \hat{u} \leadsto u :: \kappa \vdash \hat{\tau} \leadsto \tau :: \mathsf{Type}}{\hat{\Omega} \vdash \texttt{uall}\{\hat{\kappa}\}(\hat{u}.\hat{\tau}) \leadsto \texttt{all}\{\kappa\}(u.\tau) :: \mathsf{Type}} \tag{6.18j}$$

$$\frac{\hat{\Omega}, \hat{t} \leadsto t :: \mathsf{Type} \vdash \hat{\tau} \leadsto \tau :: \mathsf{Type}}{\hat{\Omega} \vdash \texttt{urec}(\hat{t}.\hat{\tau}) \leadsto \texttt{rec}(t.\tau) :: \mathsf{Type}} \tag{6.18k}$$

$$\frac{\{\hat{\Omega} \vdash \hat{\tau}_i \leadsto \tau_i :: \mathsf{Type}\}_{1 \leq i \leq n}}{\hat{\Omega} \vdash \texttt{uprod}[L](\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}) \leadsto \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) :: \mathsf{Type}} \tag{6.18l}$$

$$\frac{\{\hat{\Omega} \vdash \hat{\tau}_i \leadsto \tau_i :: \mathsf{Type}\}_{1 \leq i \leq n}}{\hat{\Omega} \vdash \texttt{usum}[L](\{i \hookrightarrow \hat{\tau}_i\}_{i \in L}) \leadsto \texttt{sum}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) :: \mathsf{Type}} \tag{6.18m}$$

$$\frac{\hat{\Omega} \vdash \hat{c} \leadsto c :: \mathsf{Type}}{\hat{\Omega} \vdash \hat{c} \leadsto c :: \mathsf{S}(c)} \tag{6.18n}$$

$$\frac{}{\hat{\Omega}, \hat{X} \leadsto X : \texttt{sig}\{\kappa\}(u.\tau) \vdash \texttt{ucon}(\hat{X}) \leadsto \texttt{con}(X) :: \kappa} \tag{6.18o}$$

Types, Expressions, Rules and Patterns

$$\frac{\hat{\Omega} \vdash \hat{\tau} \leadsto \tau :: \mathsf{Type}}{\hat{\Omega} \vdash \hat{\tau} \leadsto \tau \ \mathsf{type}} \tag{6.19}$$

Synthetic typed expression expansion

$$\frac{}{\Omega, \hat{x} \leadsto x : \tau \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{x} \leadsto x \Rightarrow \tau} \tag{6.20a}$$

$$\frac{\hat{\Omega} \vdash \hat{\tau} \leadsto \tau \ \mathsf{type} \qquad \hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \leadsto e \Leftarrow \tau}{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{uasc}\{\hat{\tau}\}(\hat{e}) \leadsto e \Rightarrow \tau} \tag{6.20b}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau \qquad \hat{\Omega}, \hat{x} \leadsto x : \tau \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e}' \leadsto e' \Rightarrow \tau'}{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{uletval}(\hat{e}; \hat{x}.\hat{e}') \leadsto \texttt{ap}(\texttt{lam}\{\tau\}(x.e'); e) \Rightarrow \tau'} \tag{6.20c}$$

$$\frac{\hat{\Omega} \vdash \hat{\tau}_1 \leadsto \tau_1 \text{ type} \qquad \hat{\Omega}, \hat{x} \leadsto x : \tau_1 \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau_2}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{ulam}\{\hat{\tau}_1\}(\hat{x}.\hat{e}) \leadsto \texttt{lam}\{\tau_1\}(x.e) \Rightarrow \texttt{parr}(\tau_1;\tau_2)} \tag{6.20d}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_1 \leadsto e_1 \Rightarrow \texttt{parr}(\tau_2;\tau) \qquad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_2 \leadsto e_2 \Leftarrow \tau_2}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{uap}(\hat{e}_1;\hat{e}_2) \leadsto \texttt{ap}(e_1;e_2) \Rightarrow \tau} \tag{6.20e}$$

$$\frac{\hat{\Omega} \vdash \hat{\kappa} \leadsto \kappa \text{ kind} \qquad \hat{\Omega}, \hat{u} \leadsto u :: \kappa \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{uclam}\{\hat{\kappa}\}(\hat{u}.\hat{e}) \leadsto \texttt{clam}\{\kappa\}(u.e) \Rightarrow \texttt{all}\{\kappa\}(u.\tau)} \tag{6.20f}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \texttt{all}\{\kappa\}(u.\tau) \qquad \hat{\Omega} \vdash \hat{c} \leadsto c :: \kappa}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{ucap}\{\hat{c}\}(\hat{e}) \leadsto \texttt{cap}\{c\}(e) \Rightarrow [c/t]\tau} \tag{6.20g}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \texttt{rec}(t.\tau)}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{uunfold}(\hat{e}) \leadsto \texttt{unfold}(e) \Rightarrow [\texttt{rec}(t.\tau)/t]\tau} \tag{6.20h}$$

$$\frac{\begin{array}{c}\hat{e} = \texttt{utpl}[L](\{i \hookrightarrow \hat{e}_i\}_{i\in L}) \qquad e = \texttt{tpl}[L](\{i \hookrightarrow e_i\}_{i\in L}) \\ \{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_i \leadsto e_i \Rightarrow \tau_i\}_{i\in L}\end{array}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i\in L})} \tag{6.20i}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \texttt{prod}[L,\ell](\{i \hookrightarrow \tau_i\}_{i\in L};\ell \hookrightarrow \tau)}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{upr}[\ell](\hat{e}) \leadsto \texttt{pr}[\ell](e) \Rightarrow \tau} \tag{6.20j}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau \qquad \{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \leadsto r_i \Rightarrow \tau \Mapsto \tau'\}_{1\le i\le n}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{umatch}[n](\hat{e};\{\hat{r}_i\}_{1\le i\le n}) \leadsto \texttt{match}[n]\{\tau'\}(e;\{r_i\}_{1\le i\le n}) \Rightarrow \tau'} \tag{6.20k}$$

TODO: these rules

$$\frac{\begin{array}{c}\Omega \vdash \hat{\rho} \leadsto \rho \text{ tsmty} \qquad \emptyset \vdash e_{\text{parse}} : \texttt{parr}(\texttt{Body};\texttt{ParseResult}(\texttt{CEExpBnd})) \\ \hat{\Psi} = \langle \mathcal{A};\Psi;\mathcal{I}\rangle \qquad \hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A}\uplus\hat{a}\hookrightarrow\texttt{defref}[a];\Psi,a\hookrightarrow\texttt{petsm}(\rho;e_{\text{parse}});\mathcal{I}\rangle;\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau'\end{array}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\rho}\ \texttt{for expressions}\ \{e_{\text{parse}}\}\ \texttt{in}\ \hat{e} \leadsto e \Rightarrow \tau'} \tag{6.20l}$$

ParseResult...
CEExpBnd...

$$\frac{\begin{array}{c}b \downarrow e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{Success} \cdot e_{\text{cand}} \qquad e_{\text{cand}} \uparrow_{\texttt{CEExp}} \grave{e} \\ \emptyset\ \emptyset \vdash^{\hat{\Delta};\hat{\Gamma};\hat{\Psi},\hat{a}\leadsto a\hookrightarrow\texttt{uetsm}(\tau;e_{\text{parse}});\hat{\Phi};b} \grave{e} \leadsto e \Leftarrow \tau\end{array}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi},\hat{a}\leadsto a\hookrightarrow\texttt{uetsm}(\tau;e_{\text{parse}});\hat{\Phi}} \hat{a}\ /b/ \leadsto e \Rightarrow \tau} \tag{6.20m}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A}\uplus\hat{a}\leadsto a;\Psi,a\hookrightarrow\texttt{uetsm}(\tau;e_{\text{parse}});\mathcal{I}\uplus\tau\hookrightarrow a\rangle;\hat{\Phi}} \hat{e} \leadsto e \Rightarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A}\uplus\hat{a}\leadsto a;\Psi,a\hookrightarrow\texttt{uetsm}(\tau;e_{\text{parse}});\mathcal{I}\rangle;\hat{\Phi}} \texttt{implicit syntax}\ \hat{a}\ \texttt{for expressions in}\ \hat{e} \leadsto e \Rightarrow \tau'} \tag{6.20n}$$

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \text{ type} \qquad \emptyset\, \emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResultPat)} \quad \hat{\Delta}\, \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi},\hat{a} \rightsquigarrow a \hookrightarrow \texttt{uptsm}(\tau;e_{\text{parse}})} \hat{e} \rightsquigarrow e \Rightarrow \tau'}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for patterns}\ \{e_{\text{parse}}\}\ \texttt{in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \tag{6.20o}$$

$$\frac{\hat{\Delta}\, \hat{\Gamma} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{uptsm}(\tau;e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle} \hat{e} \rightsquigarrow e \Rightarrow \tau'}{\hat{\Delta}\, \hat{\Gamma} \vdash_{\hat{\Psi};\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{uetsm}(\tau;e_{\text{parse}}); \mathcal{I} \rangle} \texttt{implicit syntax}\ \hat{a}\ \texttt{for patterns in}\ \hat{e} \rightsquigarrow e \Rightarrow \tau'} \tag{6.20p}$$

Analytic typed expression expansion

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \Omega \vdash \tau <: \tau'}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{6.21a}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \hat{\Omega}, \hat{x} \rightsquigarrow x : \tau \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \rightsquigarrow e' \Leftarrow \tau'}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{uletval}(\hat{e};\hat{x}.\hat{e}') \rightsquigarrow \texttt{ap(lam}\{\tau\}\texttt{(}x.e'\texttt{)};e\texttt{)} \Leftarrow \tau'} \tag{6.21b}$$

$$\frac{\hat{\Omega}, \hat{x} \rightsquigarrow x : \tau_1 \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau_2}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{uanalam}(\hat{x}.\hat{e}) \rightsquigarrow \texttt{lam}\{\tau_1\}\texttt{(}x.e\texttt{)} \Leftarrow \texttt{parr(}\tau_1;\tau_2\texttt{)}} \tag{6.21c}$$

$$\frac{\hat{\Delta}, \hat{t} \rightsquigarrow t \text{ type}\ \hat{\Gamma} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{utlam}(\hat{t}.\hat{e}) \rightsquigarrow \texttt{tlam(}t.e\texttt{)} \Leftarrow \texttt{all(}t.\tau\texttt{)}} \tag{6.21d}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow [\texttt{rec(}t.\tau\texttt{)}/t]\tau}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{ufold(}\hat{e}\texttt{)} \rightsquigarrow \texttt{fold}\{t.\tau\}\texttt{(}e\texttt{)} \Leftarrow \texttt{rec(}t.\tau\texttt{)}} \tag{6.21e}$$

$$\frac{\hat{e} = \texttt{utpl}[L]\texttt{(}\{i \hookrightarrow \hat{e}_i\}_{i \in L}\texttt{)} \qquad e = \texttt{tpl}[L]\texttt{(}\{i \hookrightarrow e_i\}_{i \in L}\texttt{)} \quad \{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}_i \rightsquigarrow e_i \Leftarrow \tau_i\}_{i \in L}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \texttt{prod}[L]\texttt{(}\{i \hookrightarrow \tau_i\}_{i \in L}\texttt{)}} \tag{6.21f}$$

$$\frac{\hat{e} = \texttt{uin}[\ell]\texttt{(}\hat{e}'\texttt{)} \qquad e = \texttt{in}[L,\ell][\ell]\{\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau\}\texttt{(}e'\texttt{)} \quad \hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e}' \rightsquigarrow e' \Leftarrow \tau}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \texttt{sum}[L,\ell]\texttt{(}\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau\texttt{)}} \tag{6.21g}$$

$$\frac{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau \qquad \{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \hat{r}_i \rightsquigarrow r_i \Leftarrow \tau \Mapsto \tau'\}_{1 \le i \le n}}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{umatch}[n]\texttt{(}\hat{e};\{\hat{r}_i\}_{1 \le i \le n}\texttt{)} \rightsquigarrow \texttt{match}[n]\{\tau'\}\texttt{(}e;\{r_i\}_{1 \le i \le n}\texttt{)} \Leftarrow \tau'} \tag{6.21h}$$

TODO: revise the following

$$\frac{\hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau \text{ type} \qquad \emptyset\, \emptyset \vdash e_{\text{parse}} : \texttt{parr(Body;ParseResultExp)} \quad \hat{\Delta}\, \hat{\Gamma} \vdash_{\hat{\Psi},\hat{a} \rightsquigarrow a \hookrightarrow \texttt{uetsm}(\tau;e_{\text{parse}});\hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Omega} \vdash_{\hat{\Psi};\hat{\Phi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for expressions}\ \{e_{\text{parse}}\}\ \texttt{in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{6.21i}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Psi, a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}}); \mathcal{I} \rangle; \hat{\Phi}} \texttt{implicit}\ \texttt{syntax}\ \hat{a}\ \texttt{for}\ \texttt{expressions}\ \texttt{in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{6.21j}$$

$$\frac{\begin{array}{c} b \downarrow e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{Success} \cdot e_{\text{cand}} \qquad e_{\text{cand}} \uparrow_{\mathsf{CEExp}} \grave{e} \\ \emptyset\ \emptyset \vdash^{\hat{\Delta}; \hat{\Gamma}; \langle \mathcal{A}; \Psi, a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle; \hat{\Phi}; b} \grave{e} \rightsquigarrow e \Leftarrow \tau \end{array}}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\langle \mathcal{A}; \Psi, a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle; \hat{\Phi}} \texttt{uelit}[b] \rightsquigarrow e \Leftarrow \tau} \tag{6.21k}$$

$$\frac{\begin{array}{c} \hat{\Delta} \vdash \hat{\tau} \rightsquigarrow \tau\ \mathsf{type} \qquad \emptyset\ \emptyset \vdash e_{\text{parse}} : \texttt{parr}(\texttt{Body}; \texttt{ParseResultPat}) \\ \hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \hat{\Phi}, \hat{a} \rightsquigarrow a \hookrightarrow \texttt{uptsm}(\tau; e_{\text{parse}})} \hat{e} \rightsquigarrow e \Leftarrow \tau' \end{array}}{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{syntax}\ \hat{a}\ \texttt{at}\ \hat{\tau}\ \texttt{for}\ \texttt{patterns}\ \{e_{\text{parse}}\}\ \texttt{in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{6.21l}$$

$$\frac{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{uptsm}(\tau; e_{\text{parse}}); \mathcal{I} \uplus \tau \hookrightarrow a \rangle} \hat{e} \rightsquigarrow e \Leftarrow \tau'}{\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}; \langle \mathcal{A} \uplus \hat{a} \rightsquigarrow a; \Phi, a \hookrightarrow \texttt{uetsm}(\tau; e_{\text{parse}}); \mathcal{I} \rangle} \texttt{implicit}\ \texttt{syntax}\ \hat{a}\ \texttt{for}\ \texttt{patterns}\ \texttt{in}\ \hat{e} \rightsquigarrow e \Leftarrow \tau'} \tag{6.21m}$$

Synthetic rule expansion

$$\frac{\begin{array}{c} \hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega \rangle \\ \Omega \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv \langle \emptyset; \mathcal{G}'; \emptyset; \Omega' \rangle \qquad \langle \mathcal{D}; \mathcal{G} \uplus \mathcal{G}'; \mathcal{M}; \Omega \cup \Omega' \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Rightarrow \tau' \end{array}}{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{urule}(\hat{p}.\hat{e}) \rightsquigarrow \texttt{rule}(p.e) \Rightarrow \tau \Mapsto \tau'} \tag{6.22}$$

Analytic rule expansion

$$\frac{\begin{array}{c} \hat{\Omega} = \langle \mathcal{D}; \mathcal{G}; \mathcal{M}; \Omega \rangle \\ \Omega \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \tau \dashv \langle \emptyset; \mathcal{G}'; \emptyset; \Omega' \rangle \qquad \langle \mathcal{D}; \mathcal{G} \uplus \mathcal{G}'; \mathcal{M}; \Omega \cup \Omega' \rangle \vdash_{\hat{\Psi}; \hat{\Phi}} \hat{e} \rightsquigarrow e \Leftarrow \tau' \end{array}}{\hat{\Omega} \vdash_{\hat{\Psi}; \hat{\Phi}} \texttt{urule}(\hat{p}.\hat{e}) \rightsquigarrow \texttt{rule}(p.e) \Leftarrow \tau \Mapsto \tau'} \tag{6.23}$$

Typed pattern expansion

$$\frac{}{\Omega \vdash_{\hat{\Phi}} \hat{x} \rightsquigarrow x : \tau \dashv \langle \emptyset; \hat{x} \rightsquigarrow x; \emptyset; x : \tau \rangle} \tag{6.24a}$$

$$\frac{}{\Omega \vdash_{\hat{\Phi}} \texttt{uwildp} \rightsquigarrow \texttt{wildp} : \tau \dashv \langle \emptyset; \emptyset; \emptyset; \emptyset \rangle} \tag{6.24b}$$

$$\frac{\Omega \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : [\texttt{rec}(t.\tau)/t]\tau \dashv \hat{\Omega}}{\Omega \vdash_{\hat{\Phi}} \texttt{ufoldp}(\hat{p}) \rightsquigarrow \texttt{foldp}(p) : \texttt{rec}(t.\tau) \dashv \hat{\Omega}} \tag{6.24c}$$

$$\frac{\begin{array}{c} \hat{p} = \texttt{utplp}[L](\{i \hookrightarrow \hat{p}_i\}_{i \in L}) \qquad p = \texttt{tplp}[L](\{i \hookrightarrow p_i\}_{i \in L}) \\ \{\Omega \vdash_{\hat{\Phi}} \hat{p}_i \rightsquigarrow p_i : \tau_i \dashv \hat{\Omega}_i\}_{i \in L} \end{array}}{\Omega \vdash_{\hat{\Phi}} \hat{p} \rightsquigarrow p : \texttt{prod}[L](\{i \hookrightarrow \tau_i\}_{i \in L}) \dashv \cup_{i \in L} \hat{\Omega}_i} \tag{6.24d}$$

$$\frac{\Omega \vdash_{\hat{\Phi}} \hat{p} \leadsto p : \tau \dashv\!\mid \hat{\Omega}}{\Omega \vdash_{\hat{\Phi}} \mathtt{uinp}[\ell](\hat{p}) \leadsto \mathtt{inp}[\ell](p) : \mathtt{sum}[L,\ell](\{i \hookrightarrow \tau_i\}_{i \in L}; \ell \hookrightarrow \tau) \dashv\!\mid \hat{\Omega}} \tag{6.24e}$$

TODO: revise these The following rule governs upTSM application. It is written identically to Rule (4.6f).

$$\frac{b \downarrow e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \mathtt{Success} \cdot e_{\text{cand}} \qquad e_{\text{cand}} \uparrow_{\mathsf{CEPat}} \grave{p} \qquad \vdash^{\Delta; \hat{\Phi}, \hat{a} \leadsto a \hookrightarrow \mathtt{uptsm}(\tau; e_{\text{parse}}); b} \grave{p} \leadsto p : \tau \dashv\!\mid^{\hat{\Omega}}}{\Delta \vdash_{\hat{\Phi}, \hat{a} \leadsto a \hookrightarrow \mathtt{uptsm}(\tau; e_{\text{parse}})} \hat{a}\ /b/ \leadsto p : \tau \dashv\!\mid \hat{\Omega}} \tag{6.24f}$$

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 (6.24f).

$$\frac{b \downarrow e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \mathtt{Success} \cdot e_{\text{cand}} \qquad e_{\text{cand}} \uparrow_{\mathsf{CEPat}} \grave{p} \qquad \vdash^{\Delta; \langle \mathcal{A}; \Phi, a \hookrightarrow \mathtt{uptsm}(\tau; e_{\text{parse}}); \mathcal{I}, \tau \hookrightarrow a \rangle; b} \grave{p} \leadsto p : \tau \dashv\!\mid^{\hat{\Omega}}}{\Delta \vdash_{\langle \mathcal{A}; \Phi, a \hookrightarrow \mathtt{uptsm}(\tau; e_{\text{parse}}); \mathcal{I}, \tau \hookrightarrow a \rangle} /b/ \leadsto p : \tau \dashv\!\mid \hat{\Omega}} \tag{6.24g}$$

Unexpanded Signatures and Module Expressions

TODO: do these

TSM Types and Expressions

TODO: judgement listing TSM type formation

$$\frac{\Omega \vdash \tau \ \mathsf{type}}{\Omega \vdash \mathtt{type}(\tau) \ \mathsf{tsmty}} \tag{6.25a}$$

$$\frac{\Omega, t :: \mathsf{Type} \vdash \rho \ \mathsf{tsmty}}{\Omega \vdash \mathtt{alltypes}(t.\rho) \ \mathsf{tsmty}} \tag{6.25b}$$

$$\frac{\Omega \vdash \sigma \ \mathsf{sig} \qquad \Omega, X : \sigma \vdash \rho \ \mathsf{tsmty}}{\Omega \vdash \mathtt{allmods}\{\sigma\}(X.\rho) \ \mathsf{tsmty}} \tag{6.25c}$$

Unexpanded TSM type expansion

$$\frac{\hat{\Omega} \vdash \hat{\tau} \leadsto \tau \ \mathsf{type}}{\hat{\Omega} \vdash \mathtt{utype}(\hat{\tau}) \leadsto \mathtt{type}(\tau) \ \mathsf{tsmty}} \tag{6.26a}$$

$$\frac{\hat{\Omega}, \hat{t} \leadsto t :: \mathsf{Type} \vdash \hat{\rho} \leadsto \rho \ \mathsf{tsmty}}{\hat{\Omega} \vdash \mathtt{ualltypes}(\hat{t}.\hat{\rho}) \leadsto \mathtt{alltypes}(t.\rho) \ \mathsf{tsmty}} \tag{6.26b}$$

$$\frac{\hat{\Omega} \vdash \hat{\sigma} \leadsto \sigma \ \mathsf{sig} \qquad \hat{\Omega}, \hat{X} \leadsto X : \sigma \vdash \hat{\rho} \leadsto \rho \ \mathsf{tsmty}}{\hat{\Omega} \vdash \mathtt{uallmods}\{\hat{\sigma}\}(\hat{X}.\hat{\rho}) \leadsto \mathtt{allmods}\{\sigma\}(X.\rho) \ \mathsf{tsmty}} \tag{6.26c}$$

6.2.6 Metatheory

Chapter 7

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.

7.1 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.

7.1.1 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. 2.2.7):

```
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)
```

7.1.2 Parser Generators

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

7.2 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.

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

Chapter 8

Discussion & Future Directions

8.1 Interesting Applications

Most of the examples in Sec. 2.2 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

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$_{\textbf{TSL}}$

A formal specification of TSLs in a language that supports only non-parametric datatypes is available in a paper published in ECOOP 2014 [31].

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 6 and Chapter 5 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 5 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 [27]) 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.

Appendix A

Dependent Labeled Product Kinds

$$\frac{\{\Omega \vdash \Delta \cup \{u_{i,j} :: \kappa_j\}_{1\leq j<i}\ \mathsf{kind}\Gamma\kappa_i\}_{1\leq i\leq n}}{\Omega \vdash \mathtt{dprod}[n;\{\ell_i\}_{1\leq i\leq n}](\{\{u_{i,j}\}_{1\leq j<i}.\kappa_i\}_{1\leq i\leq n})\ \mathsf{kind}} \tag{A.1}$$

$$\frac{\{\Omega \vdash \Delta \cup \{u_{i,j} :: \kappa_j\}_{1\leq j<i} \equiv \Gamma\kappa_i\kappa'_i\}_{1\leq i\leq n}}{\Omega \vdash \mathtt{dprod}[n;\{\ell_i\}_{1\leq i\leq n}](\{\{u_{i,j}\}_{1\leq j<i}.\kappa_i\}_{1\leq i\leq n}) \equiv \mathtt{dprod}[n;\{\ell_i\}_{1\leq i\leq n}](\{\{u_{i,j}\}_{1\leq j<i}.\kappa'_i\}_{1\leq i\leq n})} \tag{A.2}$$

$$\frac{\{\Omega \vdash \Delta \cup \{u_{i,j} :: \kappa_j\}_{1\leq j<i} <:: \Gamma\kappa_i\kappa'_i\}_{1\leq i\leq n}}{\Omega \vdash \mathtt{dprod}[n;\{\ell_i\}_{1\leq i\leq n}](\{\{u_{i,j}\}_{1\leq j<i}.\kappa_i\}_{1\leq i\leq n}) <:: \mathtt{dprod}[n;\{\ell_i\}_{1\leq i\leq n}](\{\{u_{i,j}\}_{1\leq j<i}.\kappa'_i\}_{1\leq i\leq n})} \tag{A.3}$$

$$\frac{\{\Omega \vdash \Delta \cup \{u_{i,j} :: \mathtt{S}(c_j)\}_{1\leq j<i} :: \Gamma c_i\kappa_i\}_{1\leq i\leq n}}{\Omega \vdash \mathtt{dtpl}[n;\{\ell_i\}_{1\leq i\leq n}](\{\{u_{i,j}\}_{1\leq j<i}.c_i\}_{1\leq i\leq n}) :: \mathtt{dprod}[n;\{\ell_i\}_{1\leq i\leq n}](\{\{u_{i,j}\}_{1\leq j<i}.\kappa_i\}_{1\leq i\leq n})} \tag{A.4}$$

$$\frac{\Omega \vdash c :: \mathtt{dprod}[n'+1+n'';\{\ell'_i\}_{1\leq i\leq n'},\ell,\{\ell''_i\}_{1\leq i\leq n''}](\{\{u'_{i,j}\}_{1\leq j<i}.\kappa'_i\}_{1\leq i\leq n'};\{u_j\}_{1\leq j\leq n'}.\kappa;\{\{u''_{i,j}\}_{1\leq j<i}.\kappa''_i\}_{1\leq i\leq n''})}{\Omega \vdash \mathtt{prj}[\ell](c) :: [\{\mathtt{prj}[\ell'_j](c)/u_j\}_{1\leq j\leq n'}]\kappa} \tag{A.5}$$

$$\frac{\begin{array}{c} c = \mathtt{dtpl}[n;\{\ell_i\}_{1\leq i\leq n}](\{\{u_{i,j}\}_{1\leq j<i}.c_i\}_{1\leq i\leq n}) \qquad c' = \mathtt{dtpl}[n;\{\ell_i\}_{1\leq i\leq n}](\{\{u_{i,j}\}_{1\leq j<i}.c'_i\}_{1\leq i\leq n}) \\ \{\Omega \vdash \Delta \cup \{u_{i,j} :: \mathtt{S}(\kappa_j)\}_{1\leq j<i} \equiv \Gamma :: c_ic'_i\kappa_i\}_{1\leq i\leq n} \end{array}}{\Omega \vdash c \equiv c' :: \mathtt{dprod}[n;\{\ell_i\}_{1\leq i\leq n}](\{\{u_{i,j}\}_{1\leq j<i}.\kappa_i\}_{1\leq i\leq n})} \tag{A.6}$$

$$\frac{\Omega \vdash c \equiv c' :: \mathtt{dprod}[n'+1+n'';\{\ell'_i\}_{1\leq i\leq n'},\ell,\{\ell''_i\}_{1\leq i\leq n''}](\{\{u'_{i,j}\}_{1\leq j<i}.\kappa'_i\}_{1\leq i\leq n'};\{u_j\}_{1\leq j\leq n'}.\kappa;\{\{u''_{i,j}\}_{1\leq j<i}.\kappa''_i\}_{1\leq i\leq n''})}{\Omega \vdash \mathtt{prj}[\ell](c) \equiv \mathtt{prj}[\ell](c') :: \kappa} \tag{A.7}$$

$$\frac{\begin{array}{c} c = \mathtt{dtpl}[n'+1+n'';\{\ell'_i\}_{1\leq i\leq n'},\ell,\{\ell''_i\}_{1\leq i\leq n''}](\{\{u'_{i,j}\}_{1\leq j<i}.c'_i\}_{1\leq i\leq n'};\{u_j\}_{1\leq j\leq n}.c_\ell;\{\{u''_{i,j}\}_{1\leq j<i}.c''_i\}_{1\leq i\leq n''}) \\ \Omega \vdash c :: \mathtt{dprod}[n'+1+n'';\{\ell'_i\}_{1\leq i\leq n'},\ell,\{\ell''_i\}_{1\leq i\leq n''}](\{\{u'_{i,j}\}_{1\leq j<i}.\kappa'_i\}_{1\leq i\leq n'};\{u_j\}_{1\leq j\leq n'}.\kappa;\{\{u''_{i,j}\}_{1\leq j<i}.\kappa''_i\}_{1\leq i\leq n''}) \end{array}}{\Omega \vdash \mathtt{prj}[\ell](c) \equiv [\{\mathtt{prj}[\ell'_j](c)/u_j\}_{1\leq j\leq n'}]c_\ell :: [\{\mathtt{prj}[\ell'_j](c)/u_j\}_{1\leq j\leq n'}]\kappa} \tag{A.8}$$

Bibliography

TODO (Later): List conference abbreviations.
TODO (Later): Remove extraneous nonsense from entries.

[1] [Rust] Macros. `https://doc.rust-lang.org/book/macros.html`. Retrieved Nov. 3, 2015. 3.1.2

[2] The Visible Compiler. `http://www.smlnj.org/doc/Compiler/pages/compiler.html`. 2.3.3

[3] Information technology portable operating system interface (posix) base specifications, issue 7. *Information technology Portable Operating System Interface (POSIX) Base Specifications, Issue 7*. 2.2.1

[4] OWASP Top 10 2013. `https://www.owasp.org/index.php/Top_10_2013-Top_10`, 2013. 3

[5] 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`. 2.3.3

[6] 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`. 2.3.1

[7] 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. 2.3.3

[8] Adam Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 122–133. ACM, 2010. ISBN 978-1-4503-0019-3. URL `http://doi.acm.org/10.1145/1806596.1806612`. 1.1

[9] 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`. 1.1

[10] Tom Christiansen, Brian D. Foy, and Larry Wall. *Programming Perl - Unmatched power for text processing and scripting: covers Version 5.14, 4th Edition*. O'Reilly, 2012.

ISBN 978-0-596-00492-7. URL http://www.oreilly.de/catalog/9780596004927/index.html. 2.2.1

[11] Sebastian Erdweg and Felix Rieger. A framework for extensible languages. In *GPCE '13*, pages 3–12, 2013. 1.1, 2.3.2

[12] Sebastian Erdweg, Tillmann Rendel, Christian Kastner, and Klaus Ostermann. SugarJ: Library-based syntactic language extensibility. In *OOPSLA '11*, pages 187–188, 2011. 1.1

[13] 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. 1.1, 1.1.1, 2.3.2

[14] 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. 2.3.3

[15] 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. 2.3.2

[16] 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. 1.3

[17] 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, 2.1, 6.1.2

[18] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012. 4.2.1, 4.2.3

[19] 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). 2.1, 2.2.1, 3.1.2, 3.2.1, 3.4, 3.2.2, 4.2.2

[20] T. P. Hart. MACRO definitions for LISP. Report A. I. MEMO 57, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, October 1963. 2.3.3

[21] David Herman. *A Theory of Typed Hygienic Macros*. PhD thesis, Northeastern University, Boston, MA, May 2010. 2.3.3

[22] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk). 4

[23] Simon L Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003. 1.1

[24] 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. 2.3.3

[25] 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. 1.1, 1.1, 2.3.2

[26] 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. 6.1.2

[27] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL http://coq.inria.fr. Version 8.0. 2.3.2, 8.6.4

[28] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997. 1.1, 1.2

[29] Odersky, Zenger, and Zenger. Colored local type inference. In *POPL: 28th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2001. 1.2

[30] 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. 1

[31] 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

[32] Cyrus Omar, Chenglong Wang, and Jonathan Aldrich. Composable and hygienic typed syntax macros. In *ACM Symposium on Applied Computing (SAC '15)*, 2015. 3.1

[33] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. 2.1, 3.2.1

[34] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, January 2000. ISSN 0164-0925. doi: 10.1145/345099.345100. URL http://doi.acm.org/10.1145/345099.345100. 1.2

[35] J. C. Reynolds. GEDANKEN - a simple typless language based on the principle of completeness and reference concept. *Comm. A.C.M.*, 13(5), May 1970. 1

[36] 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. 1.1.1, 2.3.2

[37] Denys Shabalin, Eugene Burmako, and Martin Odersky. Quasiquotes for Scala. Technical Report EPFL-REPORT-185242, 2013. 2.3.3

[38] 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

[39] Guy L Steele. *Common LISP: the language*. Digital press, 1990. 2.3.2

[40] 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. 2.2.1

[41] Todd L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University, 2004. 2.3.4

[42] Martin P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15 (4):147–161, 1994. 1.1.1

[43] Xi, Chen, and Chen. Guarded recursive datatype constructors. In *POPL '03*, 2003. 1.1.3