Modularly Programmable Syntax

Cyrus Omar

TODO: get TR number

November 25, 2015

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 concrete 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. Unfortunately, programmers cannot modularly 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 syntactic expansion while maintaining a type discipline, a hygienic binding discipline and modular 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) [34]

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, 27], 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 [41]. 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 [35], 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 [42]. 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 the following primitives:

1. **Typed syntax macros**, or **TSMs**. TSMs are applied like functions to *generalized literal forms* to 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 5.
2. **Type-specific languages**, or **TSLs**. TSLs, described in Chapter 6, further reduce syntactic cost by allowing library providers to designate a privileged TSM for each type that they introduce. Library clients can then rely on local type inference to invoke that TSM and apply its parameters implicitly. TSLs 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

[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.

from other dialects of ML that have a similar underlying type structure, like Standard ML and OCaml, in that it uses a local type inference scheme [33] (like, for example, Scala [28]) for reasons that have to do with the mechanisms described in Chapter 6. The reason we will not follow Standard ML [27] 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
- *modular 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 modular 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. 7.3.5.

Chapter 5 and Chapter 6, 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*) [32], 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) [39]. They are particularly common in domains like natural language processing and bioinformatics. The abstract syntax of regexes, r, over strings, s, is specified below:

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

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 sum type `Rx`

Values of type `Rx` are constructed by applying a label to a value of the type specified in the corresponding clause 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: An example of ML+Rx's support for spliced subexpressions.

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: An example of derived pattern syntax.

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)

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)

2.3 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 [29]. 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`:

   ```
   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

[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`.

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" [37].

   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

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

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 [26]. 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* [38] (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/
```

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 [35]. 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 [36], 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) [40]. TODO: flesh this out and connect it to previous stuff

Chapter 3

Unparameterized Expression TSMs

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 focuses on perhaps the simplest manifestation of TSMs: **unparameterized expression TSMs**, which generate expressions of a single specified type. We will then consider TSMs that generate patterns in Chapter 4 and TSMs defined over parameterized families of types in Chapter 5.

3.1 Expression TSMs By Example

We begin in this section with a "tutorial-style" introduction to unparameterized expression TSMs in VerseML. In particular, we discuss a TSM for constructing values of the recursive labeled sum type `Rx` that was defined in Figure 2.1. We then formally specify unparameterized expression TSMs with a reduced calculus, miniVerse$_{\mathsf{U}}$, 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:

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

```
'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 [30] and multipart forms [31], but for simplicity we leave these out of VerseML.

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 macro 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 macro 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. 7.3.1).

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.

```
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 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 4, 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. 7.1.1. 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 5.

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

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

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

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

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

The trade-off is that this prevents library providers from defining alternative binding forms. For example, Haskell's derived form for monadic commands (i.e. **do**-notation) supports binding the result of executing a command to a variable that is then available in the subsequent commands in a command sequence. In VerseML, this cannot be expressed in the same way. We will show an alternative formulation of Haskell's syntax for monadic commands that uses VerseML's anonymous function syntax to bind variables in Sec. 7.1.2. We will discuss mechanisms that would allow us to relax this restriction while retaining client control over variable names as future work in Sec. 7.3.8.

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 = let
    syntax $rx at Rx { (* ... *) }
  in (* $rx is in scope here *) end
in (* $rx is no longer in scope *) end
```

We will consider the question of how TSM definitions can be exported from within modules in Sec. 7.3.2.

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 6 we will consider a design where even this syntactic cost can be eliminated in certain situations.

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

3.2 miniVerse$_\mathbf{U}$

To make the intuitions developed in the previous section mathematically precise, we will now introduce a reduced calculus with support for unparameterized expression TSMs called miniVerse$_\mathbf{U}$.

3.2.1 Types and Expanded Expressions

The *inner core* of miniVerse$_\mathbf{U}$ 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 purely functional language with support for partial functions, quantification over types, recursive types, labeled product types and labeled sum types. The reader can consult *PFPL* [19] (or another text on type systems, e.g. *TAPL* [32]) for a detailed account of these constructs, or closely related variants thereof. For our purposes, it suffices to recall the following facts and definitions.

Statics of the Inner Core

The *statics of the inner core* is specified by hypothetical 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 has type τ assuming Δ and Γ

Sort			**Abstract 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
			$\texttt{prod}(\{i \hookrightarrow \tau_i\}_{i\in L})$	$\langle\{i \hookrightarrow \tau_i\}_{i\in L}\rangle$	labeled product
			$\texttt{sum}(\{i \hookrightarrow \tau_i\}_{i\in L})$	$[\{i \hookrightarrow \tau_i\}_{i\in L}]$	labeled sum
EExp	e	$::=$	x	x	variable
			$\texttt{elam}\{\tau\}(x.e)$	$\lambda x{:}\tau.e$	abstraction
			$\texttt{eap}(e;e)$	$e(e)$	application
			$\texttt{etlam}(t.e)$	$\Lambda t.e$	type abstraction
			$\texttt{etap}\{\tau\}(e)$	$e\texttt{[}\tau\texttt{]}$	type application
			$\texttt{efold}\{t.\tau\}(e)$	$\texttt{fold}(e)$	fold
			$\texttt{eunfold}(e)$	$\texttt{unfold}(e)$	unfold
			$\texttt{etpl}(\{i \hookrightarrow e_i\}_{i\in L})$	$\langle\{i \hookrightarrow e_i\}_{i\in L}\rangle$	labeled tuple
			$\texttt{epr}[\ell](e)$	$e \cdot \ell$	projection
			$\texttt{ein}[\ell]\{\{i \hookrightarrow \tau_i\}_{i\in L}\}(e)$	$\ell \cdot e$	injection
			$\texttt{ecase}(e;\{i \hookrightarrow x_i.e_i\}_{i\in L})$	$\texttt{case}\ e\ \{i \hookrightarrow x_i.e_i\}_{i\in L}$	case analysis

Figure 3.4: Syntax of types and expanded expressions, which form the inner core of miniVerse$_{\mathbf{U}}$. Metavariable x ranges over variables, t ranges over type variables, ℓ ranges over labels and L ranges over label sets. We write $\{i \hookrightarrow \tau_i\}_{i\in L}$ for finite mappings from the labels in the label set L to, in this case, types. We write $\{i \hookrightarrow \tau_i\}_{i\in L} \otimes \ell \hookrightarrow \tau$ when $\ell \notin L$ for the extension of $\{i \hookrightarrow \tau_i\}_{i\in L}$ that maps ℓ to τ. For concision, the label set is omitted when it can be inferred for a particular mapping, e.g. $\ell_1 \hookrightarrow e_1, \ell_2 \hookrightarrow e_2$ has label set $\{\ell_1, \ell_2\}$. Similarly, when we use the stylized forms $\texttt{fold}(e)$ and $\ell \cdot e$, we assume that the reader can infer the type information made explicit only in the corresponding abstract form from context.

Type formation contexts, Δ, consist of hypotheses of the form t type and *typing contexts*, Γ, consist of hypotheses of the form $x : \tau$. Syntactically, we write contexts as comma-separated sequences of hypotheses identified up to exchange and contraction.

The type formation judgement, $\Delta \vdash \tau$ type, ensures that all free type variables in τ are tracked by Δ. It is inductively defined by the following rules:

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

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

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

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

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

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

Premises written $\{\mathcal{J}_i\}_{i \in L}$ mean that for each $i \in L$, the judgement $\mathcal{J}_i$ must be derived.

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

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

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

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

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

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

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

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

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

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

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

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

The rules given above validate the following standard lemmas.

The Weakening Lemma expresses the intuition that extending a context with unused variables preserves well-formedness and typing.

Lemma 3.1 (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.

1. By rule induction on Rules (3.1).
2. By rule induction on Rules (3.2).
3. By rule induction on Rules (3.2).

□

The Substitution Lemma expresses the intuition that substitution of a type for a type variable, or an expression of the appropriate type for an expression variable, preserves well-formedness and typing.

Lemma 3.2 (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. *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. In each case, by rule induction on the derivation of the first assumption. □

The Decomposition Lemma is the converse of the Substitution Lemma.

Lemma 3.3 (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. *If* $\Delta\ [\tau'/t]\Gamma \vdash [\tau'/t]e : [\tau'/t]\tau$ *and* $\Delta \vdash \tau'$ type *then* Δ, t type $\Gamma \vdash e : \tau$.
3. *If* $\Delta\ \Gamma \vdash [e'/x]e : \tau$ *and* $\Delta\ \Gamma \vdash e' : \tau'$ *then* $\Delta\ \Gamma, x : \tau' \vdash e : \tau$.

Proof Sketch.

1. Type formation of $[\tau'/t]\tau$ does not depend on the structure of τ'.
2. The derivation of $\Delta\ [\tau'/t]\Gamma \vdash [\tau'/t]e : [\tau'/t]\tau$ does not depend on the structure of τ'.
3. Typing of $[e'/x]e$ depends only on the type of e' wherever it occurs, if at all.

□

The Regularity Lemma expresses the intuition that the type assigned to an expression under a well-formed typing context is well-formed.

Lemma 3.4 (Regularity). *If* $\Delta\ \Gamma \vdash e : \tau$ *and* $\Delta \vdash \tau_i$ type *for each assumption* $x_i : \tau_i$ *in* Γ *then* $\Delta \vdash \tau$ type.

Proof Sketch. By rule induction on Rules (3.2) and Lemma 3.2. □

Dynamics of miniVerse$_{\mathbf{U}}$

The *dynamics of* miniVerse$_{\mathbf{U}}$ is specified as a structural dynamics (a.k.a. structural operational semantics) by judgements of the following form:

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

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

Definition 3.5 (Iterated Transition). *Iterated transition,* $e \mapsto^* e'$*, is the reflexive, transitive closure of the transition judgement.*

Definition 3.6 (Evaluation). $e \Downarrow e'$ *iff* $e \mapsto^* e'$ *and* e' val.

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

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

Condition 3.7 (Canonical Forms). *If* $\vdash e : \tau$ *and* e val *then:*

1. *If* $\tau = \texttt{parr}(\tau_1;\tau_2)$ *then* $e = \texttt{elam}\{\tau_1\}(x.e')$ *and* $x : \tau_1 \vdash e' : \tau_2$.
2. *If* $\tau = \texttt{all}(t.\tau')$ *then* $e = \texttt{etlam}(t.e')$ *and* t type $\vdash e' : \tau'$.
3. *If* $\tau = \texttt{rec}(t.\tau')$ *then* $e = \texttt{efold}\{t.\tau'\}(e')$ *and* $\vdash e' : [\texttt{rec}(t.\tau')/t]\tau'$.
4. *If* $\tau = \texttt{prod}(\{i \hookrightarrow \tau_i\}_{i \in L})$ *then* $e = \texttt{etpl}(\{i \hookrightarrow e_i\}_{i \in L})$ *and for each* $i \in L$ *we have that* $\vdash e_i : \tau_i$ *and* e_i val.
5. *If* $\tau = \texttt{sum}(\{i \hookrightarrow \tau_i\}_{i \in L})$ *then* $e = \texttt{ein}[\ell]\{\{i \hookrightarrow \tau_i\}_{i \in L \setminus \ell} \otimes \ell \hookrightarrow \tau'\}(e')$ *and* $\ell \in L$ *and* $\vdash e' : \tau'$ *and* e' val.

We also require that the statics and dynamics be coherent. This is expressed in the standard way by Preservation and Progress conditions (together, these constitute the Type Safety Condition).

Condition 3.8 (Preservation). *If* $\vdash e : \tau$ *and* $e \mapsto e'$ *then* $\vdash e' : \tau$.

Condition 3.9 (Progress). *If* $\vdash e : \tau$ *then either* e val *or there exists an* e' *such that* $e \mapsto e'$.

3.2.2 Unexpanded Expressions and Typed Expansion

Programs evaluate as expanded expressions, but programmers author *unexpanded expressions*, $\hat{e}$. The syntax of unexpanded expressions is specified by the chart in Figure 3.5.

Sort			Abstract Form	Stylized Form	Description
UExp	$\hat{e}$	$::=$	x	x	variable
			$\texttt{ulam}\{\tau\}\texttt{(}x.\hat{e}\texttt{)}$	$\lambda x{:}\tau.\hat{e}$	abstraction
			$\texttt{uap(}\hat{e}\texttt{;}\hat{e}\texttt{)}$	$\hat{e}(\hat{e})$	application
			$\texttt{utlam(}t.\hat{e}\texttt{)}$	$\Lambda t.\hat{e}$	type abstraction
			$\texttt{utap}\{\tau\}\texttt{(}\hat{e}\texttt{)}$	$\hat{e}\texttt{[}\tau\texttt{]}$	type application
			$\texttt{ufold}\{t.\tau\}\texttt{(}\hat{e}\texttt{)}$	$\texttt{fold}(\hat{e})$	fold
			$\texttt{uunfold(}\hat{e}\texttt{)}$	$\texttt{unfold}(\hat{e})$	unfold
			$\texttt{utpl(}\{i \hookrightarrow \hat{e}_i\}_{i \in L}\texttt{)}$	$\langle\{i \hookrightarrow \hat{e}_i\}_{i \in L}\rangle$	labeled tuple
			$\texttt{upr[}\ell\texttt{](}\hat{e}\texttt{)}$	$\hat{e} \cdot \ell$	projection
			$\texttt{uin[}\ell\texttt{]}\{\{i \hookrightarrow \tau_i\}_{i \in L}\}\texttt{(}\hat{e}\texttt{)}$	$\ell \cdot \hat{e}$	injection
			$\texttt{ucase(}\hat{e}\texttt{;}\{i \hookrightarrow x_i.\hat{e}_i\}_{i \in L}\texttt{)}$	$\texttt{case}\ \hat{e}\ \{i \hookrightarrow x_i.\hat{e}_i\}_{i \in L}$	case analysis
			$\texttt{usyntax}\{\hat{e}\}\{\tau\}\texttt{(}a.\hat{e}\texttt{)}$	$\texttt{syntax}\ a\ \texttt{at}\ \tau\ \{\hat{e}\}\ \texttt{in}\ \hat{e}$	macro definition
			$\texttt{utsmap}[b]\texttt{[}a\texttt{]}$	$a\ \texttt{/}b\texttt{/}$	macro application

Figure 3.5: Syntax of unexpanded expressions in miniVerse$_\mathsf{U}$. Metavariable a ranges over macro names and b ranges over literal bodies. Literal bodies might contain unparsed subexpressions, so variable renaming and substitution cannot be defined in the usual manner over unexpanded expressions (i.e. they should be thought of as partially parsed abstract syntax trees, not abstract binding trees.)

Unexpanded expressions are typed and expanded simultaneously according to the *typed expansion judgement*:

Judgement Form	Description
$\Delta\ \Gamma \vdash_\Sigma \hat{e} \rightsquigarrow e : \tau$	$\hat{e}$ expands to e at type τ assuming Δ and Γ under macro environment Σ

The typed expansion judgement is inductively defined by the following rules:

$$\dfrac{}{\Delta\ \Gamma, x : \tau \vdash_\Sigma x \rightsquigarrow x : \tau} \tag{3.3a}$$

$$\dfrac{\Delta \vdash \tau\ \mathsf{type} \qquad \Delta\ \Gamma, x : \tau \vdash_\Sigma \hat{e} \rightsquigarrow e : \tau'}{\Delta\ \Gamma \vdash_\Sigma \texttt{ulam}\{\tau\}\texttt{(}x.\hat{e}\texttt{)} \rightsquigarrow \texttt{elam}\{\tau\}\texttt{(}x.e\texttt{)} : \texttt{parr(}\tau\texttt{;}\tau'\texttt{)}} \tag{3.3b}$$

$$\dfrac{\Delta\ \Gamma \vdash_\Sigma \hat{e}_1 \rightsquigarrow e_1 : \texttt{parr(}\tau\texttt{;}\tau'\texttt{)} \qquad \Delta\ \Gamma \vdash_\Sigma \hat{e}_2 \rightsquigarrow e_2 : \tau}{\Delta\ \Gamma \vdash_\Sigma \texttt{uap(}\hat{e}_1\texttt{;}\hat{e}_2\texttt{)} \rightsquigarrow \texttt{eap(}e_1\texttt{;}e_2\texttt{)} : \tau'} \tag{3.3c}$$

$$\dfrac{\Delta, t\ \mathsf{type}\ \Gamma \vdash_\Sigma \hat{e} \rightsquigarrow e : \tau}{\Delta\ \Gamma \vdash_\Sigma \texttt{utlam(}t.\hat{e}\texttt{)} \rightsquigarrow \texttt{etlam(}t.e\texttt{)} : \texttt{all(}t.\tau\texttt{)}} \tag{3.3d}$$

$$\dfrac{\Delta\ \Gamma \vdash_\Sigma \hat{e} \rightsquigarrow e : \texttt{all(}t.\tau\texttt{)} \qquad \Delta \vdash \tau'\ \mathsf{type}}{\Delta\ \Gamma \vdash_\Sigma \texttt{utap}\{\tau'\}\texttt{(}\hat{e}\texttt{)} \rightsquigarrow \texttt{etap}\{\tau'\}\texttt{(}e\texttt{)} : [\tau'/t]\tau} \tag{3.3e}$$

$$\dfrac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type} \qquad \Delta\ \Gamma \vdash_\Sigma \hat{e} \rightsquigarrow e : [\texttt{rec(}t.\tau\texttt{)}/t]\tau}{\Delta\ \Gamma \vdash_\Sigma \texttt{ufold}\{t.\tau\}\texttt{(}\hat{e}\texttt{)} \rightsquigarrow \texttt{efold}\{t.\tau\}\texttt{(}e\texttt{)} : \texttt{rec(}t.\tau\texttt{)}} \tag{3.3f}$$

$$\frac{\Delta\ \Gamma \vdash_\Sigma \hat{e} \leadsto e : \texttt{rec}(t.\tau)}{\Delta\ \Gamma \vdash_\Sigma \texttt{uunfold}(\hat{e}) \leadsto \texttt{eunfold}(e) : [\texttt{rec}(t.\tau)/t]\tau} \tag{3.3g}$$

$$\frac{\{\Delta\ \Gamma \vdash_\Sigma \hat{e}_i \leadsto e_i : \tau_i\}_{i\in L}}{\Delta\ \Gamma \vdash_\Sigma \texttt{utpl}(\{i \hookrightarrow \hat{e}_i\}_{i\in L}) \leadsto \texttt{etpl}(\{i \hookrightarrow e_i\}_{i\in L}) : \texttt{prod}(\{i \hookrightarrow \tau_i\}_{i\in L})} \tag{3.3h}$$

$$\frac{\Delta\ \Gamma \vdash_\Sigma \hat{e} \leadsto e : \texttt{prod}(\{i \hookrightarrow \tau_i\}_{i\in L} \otimes \ell \hookrightarrow \tau)}{\Delta\ \Gamma \vdash_\Sigma \texttt{upr[}\ell\texttt{]}(\hat{e}) \leadsto \texttt{epr[}\ell\texttt{]}(e) : \tau} \tag{3.3i}$$

$$\frac{\{\Delta \vdash \tau_i \ \mathsf{type}\}_{i\in L} \qquad \Delta \vdash \tau \ \mathsf{type} \qquad \Delta\ \Gamma \vdash_\Sigma \hat{e} \leadsto e : \tau}{\left\{\begin{array}{c}\Delta\ \Gamma \vdash_\Sigma \texttt{uin[}\ell\texttt{]}\{\{i \hookrightarrow \tau_i\}_{i\in L} \otimes \ell \hookrightarrow \tau\}(\hat{e}) \\ \leadsto \\ \texttt{ein[}\ell\texttt{]}\{\{i \hookrightarrow \tau_i\}_{i\in L} \otimes \ell \hookrightarrow \tau\}(e) : \texttt{sum}(\{i \hookrightarrow \tau_i\}_{i\in L} \otimes \ell \hookrightarrow \tau)\end{array}\right\}} \tag{3.3j}$$

$$\frac{\Delta\ \Gamma \vdash_\Sigma \hat{e} \leadsto e : \texttt{sum}(\{i \hookrightarrow \tau_i\}_{i\in L}) \qquad \{\Delta\ \Gamma, x_i : \tau_i \vdash_\Sigma \hat{e}_i \leadsto e_i : \tau\}_{i\in L}}{\Delta\ \Gamma \vdash_\Sigma \texttt{ucase}(\hat{e}; \{i \hookrightarrow x_i.\hat{e}_i\}_{i\in L}) \leadsto \texttt{ecase}(e; \{i \hookrightarrow x_i.e_i\}_{i\in L}) : \tau} \tag{3.3k}$$

$$\frac{\begin{array}{c}\Delta \vdash \tau \ \mathsf{type} \qquad \emptyset\ \emptyset \vdash_\emptyset \hat{e}_{\text{parse}} \leadsto e_{\text{parse}} : \texttt{parr(Body;ParseResultExp)} \\ a \notin \text{dom}(\Sigma) \qquad \Delta\ \Gamma \vdash_{\Sigma, a \hookrightarrow \texttt{syntax}(\tau; e_{\text{parse}})} \hat{e} \leadsto e : \tau'\end{array}}{\Delta\ \Gamma \vdash_\Sigma \texttt{usyntax}\{\hat{e}_{\text{parse}}\}\{\tau\}(a.\hat{e}) \leadsto e : \tau'} \tag{3.3l}$$

$$\frac{\begin{array}{c}b \downarrow e_{\text{body}} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow \texttt{Success} \cdot e_{\text{cand}} \qquad e_{\text{cand}} \uparrow_{\mathsf{CEExp}} \grave{e} \\ \emptyset\ \emptyset \vdash^{\Delta;\Gamma;\Sigma, a \hookrightarrow \texttt{syntax}(\tau; e_{\text{parse}});b} \grave{e} \leadsto e : \tau\end{array}}{\Delta\ \Gamma \vdash_{\Sigma, a \hookrightarrow \texttt{syntax}(\tau; e_{\text{parse}})} \texttt{utsmap}[b]\texttt{[}a\texttt{]} \leadsto e : \tau} \tag{3.3m}$$

Notice that each form of expanded expression (Figure 3.4) corresponds to a form of unexpanded expression (Figure 3.5). For each typing rule in Rules (3.2), there is a corresponding typed expansion rule – Rules (3.3a) through (3.3k) – where the unexpanded and expanded forms correspond. The premises also correspond – if a typing judgement appears as a premise of a typing rule, then the corresponding premise in the corresponding typed expansion rule is the corresponding typed expansion judgement. The macro environment is not extended or inspected by these rules (it is only "threaded through" them opaquely).

There are two unexpanded expression forms that do not correspond to an expanded expression form: the macro definition form, and the macro application form. The rules governing these two forms interact with the macro environment, and are the topics of the next two subsections, respectively.

3.2.3 Macro Definition

The *macro definition form*:

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

allows the programmer to introduce a macro named a at type τ with *unexpanded parse function* $\hat{e}_{\text{parse}}$ into the macro environment of $\hat{e}$. The abstract form corresponding to this stylized form is $\texttt{usyntax}\{\hat{e}_{\text{parse}}\}\{\tau\}(a.\hat{e})$. The premises of Rule (3.3l), which governs this form, can be understood as follows, in order:

1. The first premise ensures that the type annotation specifies a well-formed type, τ.
2. The second premise types and expands the *unexpanded parse function*, $\hat{e}_{\text{parse}}$, to produce the *expanded parse function*, e_{parse}. Notice that this occurs under empty contexts, i.e. parse functions cannot refer to the surrounding variable bindings. This is because parse functions are evaluated when a macro is applied during the typed expansion process (as we will discuss momentarily), not during evaluation of the program they appear within. Parse functions must be of type

   $$\mathtt{parr(Body;ParseResultExp)}$$

   where `ParseResultExp` abbreviates the following labeled sum type[1]:

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

   and `Body` and `CEExp` abbreviate types that we will characterize below.
3. Macro environments are finite mappings from macro names, a, to *expanded macro definitions*, $\mathtt{syntax}(\tau; e_{\text{parse}})$, where τ is the macro's type annotation and e_{parse} is the macro's expanded parse function. The *macro environment formation judgement*, $\Delta \vdash \Sigma$ menv, ensures that the type annotations in Σ are well-formed assuming Δ and the parse functions in Σ are of type $\mathtt{parr(Body;ParseResultExp)}$.

Judgement Form	**Description**
$\Delta \vdash \Sigma$ menv	Macro environment Σ is well-formed assuming Δ.

   This judgement is inductively defined by the following rules:

   $$\frac{}{\Delta \vdash \emptyset \ \mathsf{menv}} \tag{3.4a}$$

   $$\frac{\Delta \vdash \Sigma \ \mathsf{menv} \qquad \Delta \vdash \tau \ \mathsf{type} \qquad \emptyset\,\emptyset \vdash e_{\text{parse}} : \mathtt{parr(Body;ParseResultExp)}}{\Delta \vdash \Sigma, a \hookrightarrow \mathtt{syntax}(\tau; e_{\text{parse}}) \ \mathsf{menv}} \tag{3.4b}$$

   The third premise of Rule (3.3l) checks that there is not already a macro named a in the macro environment, Σ.
4. The fourth premise of Rule (3.3l) extends the macro environment with the newly determined expanded macro definition and proceeds to produce a type, τ', and expansion, e, for $\hat{e}$.

The conclusion of Rule (3.3l) specifies τ' and e as the type and expansion of the expression as a whole, i.e. macro definitions "disappear" when an unexpanded expression is expanded (because they specify behavior that is relevant only during typed expansion).

[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

3.2.4 Macro Application

The last form of unexpanded expression is the form for applying a macro named a to a literal form with literal body b.

$$a\ /b/$$

The stylized form for macro application, shown above, uses forward slashes as delimiters, though stylized variants of any of the literal forms specified in Figure 3.1 would be straightforward to add to the syntax table in Figure 3.5 (we omit them for concision).

The abstract form corresponding to this stylized form is $\texttt{utsmap}[b][a]$, i.e. for each literal body, b, there is an operator $\texttt{utsmap}[b]$, indexed by the macro name a and taking no arguments. Macro names are symbolic, i.e. they can be compared to one another, but unlike variables, they do not stand in for terms. Instead, they are simply references into the macro environment.

The premises of Rule (3.3m), which governs the macro application form, can be understood as follows, in order:

1. The *body encoding judgement* $b \downarrow e_{\text{body}}$ specifies a mapping from the literal body, b, to an expanded value, e_{body}, of type $\texttt{Body}$. An inverse mapping is specified by the *body decoding judgement* $e_{\text{body}} \uparrow b$.

Judgement Form	**Description**
$b \downarrow e$	b encodes to e
$e \uparrow b$	e decodes to b

   Rather than picking a particular definition of $\texttt{Body}$ and inductively defining these judgements against it, we simply state the following sufficient conditions, which establish an isomorphism between literal bodies and values of type $\texttt{Body}$.

   Condition 3.10 (Body Encoding)**.** *For every literal body b, we have that $b \downarrow e_{body}$ and $\vdash e_{body} : \texttt{Body}$ and e_{body} val.*

   Condition 3.11 (Body Decoding)**.** *If $\vdash e_{body} : \texttt{Body}$ and e_{body} val then $e_{body} \uparrow b$ for some b.*

   Condition 3.12 (Body Encoding Inverse)**.** *If $b \downarrow e_{body}$ then $e_{body} \uparrow b$.*

   Condition 3.13 (Body Decoding Inverse)**.** *If $\vdash e_{body} : \texttt{Body}$ and e_{body} val and $e_{body} \uparrow b$ then $b \downarrow e_{body}$.*

   Condition 3.14 (Body Encoding Uniqueness)**.** *If $b \downarrow e_{body}$ and $b \downarrow e'_{body}$ then $e_{body} = e'_{body}$.*

   Condition 3.15 (Body Decoding Uniqueness)**.** *If $\vdash e_{body} : \texttt{Body}$ and e_{body} val and $e_{body} \uparrow b$ and $e_{body} \uparrow b'$ then $b = b'$.*

2. The second premise applies the expanded parse function e_{parse} associated with a in the macro environment 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 $\texttt{CEExp}$ (assuming a well-formed macro environment and by the definition of evaluation, transitive application of the Preservation Assumption 4.4 and the Canonical Forms Assumption 4.3). 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 judgement $e_{\text{cand}} \uparrow_{\mathsf{CEExp}} \grave{e}$ decodes the encoding of the candidate expansion, i.e. it maps e_{cand} onto a *candidate expansion expression*, $\grave{e}$. The syntax of candidate expansion expressions and *candidate expansion types*, $\grave{\tau}$, which can occur within candidate expansion expressions, is specified in Figure 3.6. The inverse mapping is specified by the judgement $\grave{e} \downarrow_{\mathsf{CEExp}} e_{\text{cand}}$.

Judgement Form	**Description**
$e \uparrow_{\mathsf{CEExp}} \grave{e}$	e decodes to $\grave{e}$
$\grave{e} \downarrow_{\mathsf{CEExp}} e$	$\grave{e}$ encodes to e

As with type `Body`, rather than defining the type `CEExp` (and other associated types) explicitly, and these judgements inductively against these definitions, we give the following sufficient conditions. These establish an isomorphism between values of type `CEExp` and candidate expansion expressions.

Condition 3.16 (Candidate Expansion Decoding). *If* $\vdash e_{cand} : \texttt{CEExp}$ *and* e_{cand} val *then* $e_{cand} \uparrow_{\mathsf{CEExp}} \grave{e}$ *for some* $\grave{e}$.

Condition 3.17 (Candidate Expansion Encoding). *For every* $\grave{e}$, *we have* $\grave{e} \downarrow_{\mathsf{CEExp}} e_{cand}$ *such that* $\vdash e_{cand} : \texttt{CEExp}$ *and* e_{cand} val.

Condition 3.18 (Candidate Expansion Decoding Inverse). *If* $\vdash e_{cand} : \texttt{CEExp}$ *and* e_{cand} val *and* $e_{cand} \uparrow_{\mathsf{CEExp}} \grave{e}$ *then* $\grave{e} \downarrow_{\mathsf{CEExp}} e_{cand}$.

Condition 3.19 (Candidate Expansion Encoding Inverse). *If* $\grave{e} \downarrow_{\mathsf{CEExp}} e_{cand}$ *then* $\grave{e} \uparrow_{\mathsf{CEExp}} e_{cand}$.

Condition 3.20 (Candidate Expansion Decoding Uniqueness). *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* $\grave{e} = \grave{e}'$.

Condition 3.21 (Candidate Expansion Encoding Uniqueness). *If* $\grave{e} \downarrow_{\mathsf{CEExp}} e_{cand}$ *and* $\grave{e} \downarrow_{\mathsf{CEExp}} e'_{cand}$ *then* $e_{cand} = e'_{cand}$.

4. The final premise of Rule (3.3m) *validates* the candidate expansion and simultaneously generates the *final expansion*, e. This is the topic of the next subsection.

3.2.5 Candidate Expansion Validation

The *candidate expansion validation judgements* validate candidate expansion types, $\grave{\tau}$, and candidate expansion expressions, $\grave{e}$, and simultaneously generate their final expansions.

Judgement Form	**Description**
$\Delta \vdash^{\mathcal{S}} \grave{\tau} \leadsto \tau$ type	Candidate expansion type $\grave{\tau}$ expands to τ under Δ and splicing scene $\mathcal{S}$.
$\Delta\ \Gamma \vdash^{\mathcal{S}} \grave{e} \leadsto e : \tau$	Candidate expansion expression $\grave{e}$ expands to e at type τ under Δ, Γ and splicing scene $\mathcal{S}$.

Splicing scenes, $\mathcal{S}$, are of the form $\Delta; \Gamma; \Sigma; b$. They consist of the type formation context, Δ, the typing context, Γ, the macro environment, Σ, and the literal body, b, from the macro application site (cf. Rule (3.3m)).

Sort			Abstract 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
			$\mathtt{ceprod}(\{i \hookrightarrow \grave{\tau}_i\}_{i \in L})$	$\langle\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}\rangle$	labeled product
			$\mathtt{cesum}(\{i \hookrightarrow \grave{\tau}_i\}_{i \in L})$	$[\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}]$	labeled sum
			$\mathtt{cesplicedt}[m;n]$	$\mathtt{spliced}\langle m,n\rangle$	spliced
CEExp	$\grave{e}$	$::=$	x	x	variable
			$\mathtt{celam}\{\grave{\tau}\}(x.\grave{e})$	$\lambda x{:}\grave{\tau}.\grave{e}$	abstraction
			$\mathtt{ceap}(\grave{e};\grave{e})$	$\grave{e}(\grave{e})$	application
			$\mathtt{cetlam}(t.\grave{e})$	$\Lambda t.\grave{e}$	type abstraction
			$\mathtt{cetap}\{\grave{\tau}\}(\grave{e})$	$\grave{e}[\grave{\tau}]$	type application
			$\mathtt{cefold}\{t.\grave{\tau}\}(\grave{e})$	$\mathtt{fold}(\grave{e})$	fold
			$\mathtt{ceunfold}(\grave{e})$	$\mathtt{unfold}(\grave{e})$	unfold
			$\mathtt{cetpl}(\{i \hookrightarrow \grave{e}_i\}_{i \in L})$	$\langle\{i \hookrightarrow \grave{e}_i\}_{i \in L}\rangle$	labeled tuple
			$\mathtt{cepr}[\ell](\grave{e})$	$\grave{e} \cdot \ell$	projection
			$\mathtt{cein}[\ell]\{\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}\}(\grave{e})$	$\ell \cdot \grave{e}$	injection
			$\mathtt{cecase}(\grave{e};\{i \hookrightarrow x_i.\grave{e}_i\}_{i \in L})$	$\mathtt{case}\ \grave{e}\ \{i \hookrightarrow x_i.\grave{e}_i\}_{i \in L}$	case analysis
			$\mathtt{cesplicede}[m;n]$	$\mathtt{spliced}\langle m,n\rangle$	spliced

Figure 3.6: Syntax of candidate expansion types, $\grave{\tau}$ (pronounced "grave τ"), and candidate expansion expressions, $\grave{e}$ (pronounced "grave e"), in miniVerse$_{\mathbf{U}}$. Metavariables m and n range over natural numbers.

Candidate Expansion Type Validation

The *candidate expansion type validation judgement*, $\Delta \vdash^{\mathcal{S}} \grave{\tau} \leadsto \tau\ \mathsf{type}$, is inductively defined by the following rules:

$$\frac{}{\Delta, t\ \mathsf{type} \vdash^{\mathcal{S}} t \leadsto t\ \mathsf{type}} \tag{3.5a}$$

$$\frac{\Delta \vdash^{\mathcal{S}} \grave{\tau}_1 \leadsto \tau_1\ \mathsf{type} \qquad \Delta \vdash^{\mathcal{S}} \grave{\tau}_2 \leadsto \tau_2\ \mathsf{type}}{\Delta \vdash^{\mathcal{S}} \texttt{ceparr}(\grave{\tau}_1;\grave{\tau}_2) \leadsto \texttt{parr}(\tau_1;\tau_2)\ \mathsf{type}} \tag{3.5b}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash^{\mathcal{S}} \grave{\tau} \leadsto \tau\ \mathsf{type}}{\Delta \vdash^{\mathcal{S}} \texttt{ceall}(t.\grave{\tau}) \leadsto \texttt{all}(t.\tau)\ \mathsf{type}} \tag{3.5c}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash^{\mathcal{S}} \grave{\tau} \leadsto \tau\ \mathsf{type}}{\Delta \vdash^{\mathcal{S}} \texttt{cerec}(t.\grave{\tau}) \leadsto \texttt{rec}(t.\tau)\ \mathsf{type}} \tag{3.5d}$$

$$\frac{\{\Delta \vdash^{\mathcal{S}} \grave{\tau}_i \leadsto \tau_i\ \mathsf{type}\}_{i \in L}}{\Delta \vdash^{\mathcal{S}} \texttt{ceprod}(\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}) \leadsto \texttt{prod}(\{i \hookrightarrow \tau_i\}_{i \in L})\ \mathsf{type}} \tag{3.5e}$$

$$\frac{\{\Delta \vdash^{\mathcal{S}} \grave{\tau}_i \leadsto \tau_i\ \mathsf{type}\}_{i \in L}}{\Delta \vdash^{\mathcal{S}} \texttt{cesum}(\{i \hookrightarrow \grave{\tau}_i\}_{i \in L}) \leadsto \texttt{sum}(\{i \hookrightarrow \tau_i\}_{i \in L})\ \mathsf{type}} \tag{3.5f}$$

$$\frac{\mathsf{parseTyp}(\mathsf{subseq}(b;m;n)) = \tau \qquad \Delta \cap \Delta_S = \emptyset \qquad \Delta_S \vdash \tau\ \mathsf{type}}{\Delta \vdash^{\Gamma_S;\Delta_S;\Sigma_S;b} \texttt{cesplicedt}[m;n] \leadsto \tau\ \mathsf{type}} \tag{3.5g}$$

Each form of type, τ, corresponds to a form of candidate expansion type, $\grave{\tau}$ (compare Figures 3.4 and 3.6). For each type formation rule in Rules (3.1), there is a corresponding candidate expansion type validation rule – Rules (3.5a) to (3.5f) – where the candidate expansion type and the final expansion correspond. The premises also correspond.

The only form of candidate expansion type that does not correspond to a form of type is $\texttt{cesplicedt}[m;n]$, which is a *reference to a spliced type*, i.e. it indicates that a type should be parsed out from the literal body, b, beginning at position m and ending at position n. Rule (3.5g) governs this form. The first premise extracts the indicated subsequence of b (using the metafunction $\mathsf{subseq}(b;m;n)$) and parses it (using the metafunction $\mathsf{parseTyp}(b)$) to produce the spliced type, τ. We assume sensible definitions of these metafunctions.

The third premise of Rule (3.5g) checks that τ is well-formed under the application site type formation context, Δ_S. The second premise requires that the expansion's type formation context, Δ, be disjoint from the application site type formation context, Δ_S. This can always be achieved by alpha-varying the candidate expansion type that the reference to the spliced type appears within. Such a change in bound variable names cannot "leak into" spliced types because the hypotheses in Δ are not made available to the spliced type τ. This achieves the *expansion independent splicing* property described in Sec. 3.1.3 for type variables. Rule (3.5g) is the only rule where Δ_S appears, so this also achieves the *context-independent expansion* property described in Sec. 3.1.3 for type variables.

Candidate Expansion Expression Validation

The *candidate expansion expression validation judgement*, $\Delta\ \Gamma \vdash^{\mathcal{S}} \grave{e} \leadsto e : \tau$, is defined mutually inductively with Rules (3.3), because a typed expansion judgement appears as a premise in Rule (3.6l) below, and a candidate expansion expression validation judgement appears as a premise in Rule (3.3m) above.

$$\frac{}{\Delta\ \Gamma, x : \tau \vdash^{\mathcal{S}} x \leadsto x : \tau} \tag{3.6a}$$

$$\frac{\Delta \vdash^{\mathcal{S}} \grave{\tau} \leadsto \tau\ \mathsf{type} \qquad \Delta\ \Gamma, x : \tau \vdash^{\mathcal{S}} \grave{e} \leadsto e : \tau'}{\Delta\ \Gamma \vdash^{\mathcal{S}} \mathtt{celam}\{\grave{\tau}\}(x.\grave{e}) \leadsto \mathtt{elam}\{\tau\}(x.e) : \mathtt{parr}(\tau;\tau')} \tag{3.6b}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathcal{S}} \grave{e}_1 \leadsto e_1 : \mathtt{parr}(\tau;\tau') \qquad \Delta\ \Gamma \vdash^{\mathcal{S}} \grave{e}_2 \leadsto e_2 : \tau}{\Delta\ \Gamma \vdash^{\mathcal{S}} \mathtt{ceap}(\grave{e}_1;\grave{e}_2) \leadsto \mathtt{eap}(e_1;e_2) : \tau'} \tag{3.6c}$$

$$\frac{\Delta, t\ \mathsf{type}\ \Gamma \vdash^{\mathcal{S}} \grave{e} \leadsto e : \tau}{\Delta\ \Gamma \vdash^{\mathcal{S}} \mathtt{cetlam}(t.\grave{e}) \leadsto \mathtt{etlam}(t.e) : \mathtt{all}(t.\tau)} \tag{3.6d}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathcal{S}} \grave{e} \leadsto e : \mathtt{all}(t.\tau) \qquad \Delta \vdash^{\mathcal{S}} \grave{\tau}' \leadsto \tau'\ \mathsf{type}}{\Delta\ \Gamma \vdash^{\mathcal{S}} \mathtt{cetap}\{\grave{\tau}'\}(\grave{e}) \leadsto \mathtt{etap}\{\tau'\}(e) : [\tau'/t]\tau} \tag{3.6e}$$

$$\frac{\Delta, t\ \mathsf{type} \vdash^{\mathcal{S}} \grave{\tau} \leadsto \tau\ \mathsf{type} \qquad \Delta\ \Gamma \vdash^{\mathcal{S}} \grave{e} \leadsto e : [\mathtt{rec}(t.\tau)/t]\tau}{\Delta\ \Gamma \vdash^{\mathcal{S}} \mathtt{cefold}\{t.\grave{\tau}\}(\grave{e}) \leadsto \mathtt{efold}\{t.\tau\}(e) : \mathtt{rec}(t.\tau)} \tag{3.6f}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathcal{S}} \grave{e} \leadsto e : \mathtt{rec}(t.\tau)}{\Delta\ \Gamma \vdash^{\mathcal{S}} \mathtt{ceunfold}(\grave{e}) \leadsto \mathtt{eunfold}(e) : [\mathtt{rec}(t.\tau)/t]\tau} \tag{3.6g}$$

$$\frac{\{\Delta\ \Gamma \vdash^{\mathcal{S}} \grave{e}_i \leadsto e_i : \tau_i\}_{i\in L}}{\Delta\ \Gamma \vdash^{\mathcal{S}} \mathtt{cetpl}(\{i \hookrightarrow \grave{e}_i\}_{i\in L}) \leadsto \mathtt{etpl}(\{i \hookrightarrow e_i\}_{i\in L}) : \mathtt{prod}(\{i \hookrightarrow \tau_i\}_{i\in L})} \tag{3.6h}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathcal{S}} \grave{e} \leadsto e : \mathtt{prod}(\{i \hookrightarrow \tau_i\}_{i\in L} \otimes \ell \hookrightarrow \tau)}{\Delta\ \Gamma \vdash^{\mathcal{S}} \mathtt{cepr}[\ell](\grave{e}) \leadsto \mathtt{epr}[\ell](e) : \tau} \tag{3.6i}$$

$$\frac{\{\Delta \vdash^{\mathcal{S}} \grave{\tau}_i \leadsto \tau_i\ \mathsf{type}\}_{i\in L} \qquad \Delta \vdash^{\mathcal{S}} \grave{\tau} \leadsto \tau\ \mathsf{type} \qquad \Delta\ \Gamma \vdash^{\mathcal{S}} \grave{e} \leadsto e : \tau}{\left\{\begin{array}{c} \Delta\ \Gamma \vdash_{\Sigma} \mathtt{cein}[\ell]\{\{i \hookrightarrow \grave{\tau}_i\}_{i\in L} \otimes \ell \hookrightarrow \grave{\tau}\}(\grave{e}) \\ \leadsto \\ \mathtt{ein}[\ell]\{\{i \hookrightarrow \tau_i\}_{i\in L} \otimes \ell \hookrightarrow \tau\}(e) : \mathtt{sum}(\{i \hookrightarrow \tau_i\}_{i\in L} \otimes \ell \hookrightarrow \tau) \end{array}\right\}} \tag{3.6j}$$

$$\frac{\Delta\ \Gamma \vdash^{\mathcal{S}} \grave{e} \leadsto e : \mathtt{sum}(\{i \hookrightarrow \tau_i\}_{i\in L}) \qquad \{\Delta\ \Gamma, x_i : \tau_i \vdash^{\mathcal{S}} \hat{e}_i \leadsto e_i : \tau\}_{i\in L}}{\Delta\ \Gamma \vdash^{\mathcal{S}} \mathtt{cecase}(\grave{e};\{i \hookrightarrow x_i.\grave{e}_i\}_{i\in L}) \leadsto \mathtt{ecase}(e;\{i \hookrightarrow x_i.e_i\}_{i\in L}) : \tau} \tag{3.6k}$$

$$\frac{\begin{array}{c} \mathsf{parseUExp}(\mathsf{subseq}(b;m;n)) = \hat{e} \\ \Delta \cap \Delta_S = \emptyset \qquad \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma_S) = \emptyset \qquad \Delta_S\ \Gamma_S \vdash_{\Sigma_S} \hat{e} \leadsto e : \tau \end{array}}{\Delta\ \Gamma \vdash^{\Delta_S;\Gamma_S;\Sigma_S;b} \mathtt{cesplicede}[m;n] \leadsto e : \tau} \tag{3.6l}$$

Each form of expanded expression, e, corresponds to a form of candidate expansion expression, $\grave{e}$ (compare Figure 3.4 and Figure 3.6). For each typing rule in Rules 3.2, there is a corresponding candidate expansion expression validation rule – Rules (3.6a) to (3.6k) – where the candidate expansion expression and expanded expression correspond. The premises also correspond.

The only form of candidate expansion expression that does not correspond to a form of expanded expression 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, b, beginning at position m and ending at position n. Rule (3.6l) governs this form. The first premise extracts the indicated subsequence of b (using the metafunction $\mathsf{subseq}(b; m; n)$) and parses it (using the metafunction $\mathsf{parseUExp}(b)$) to produce the spliced unexpanded expression, $\hat{e}$. We assume sensible definitions of these metafunctions.

The final premise of Rule (3.6l) types and expands the spliced unexpanded expression $\hat{e}$ under the application site contexts, Δ_S and Γ_S, and macro environment, Σ_S. The second premise requires that the expansion's type formation context, Δ, be disjoint from the application site type formation context, Δ_S. Similarly, the third premise requires that the expansion's typing context, Γ, be disjoint from the application site typing context, Γ_S. These requirements can always be satisfied by alpha-varying the candidate expansion expression that the reference to the spliced unexpanded expression appears within. Such a change in bound variable names cannot "leak into" spliced unexpanded expressions because the hypotheses in Δ and Γ are not available to the spliced unexpanded expression $\hat{e}$. This achieves the *expansion independent splicing* property described in Sec. 3.1.3 for variables and type variables. Rule (3.6l) is the only rule where Δ_S, Γ_S and Σ_S appear. This achieves the context-independent expansion property described in Sec. 3.1.3 for variables and type variables.

Candidate expansions cannot themselves define or apply TSMs. This simplifies our metatheory. We discuss relaxing this restriction in Sec. 7.3.3.

3.2.6 Metatheory

For the judgements we have specified to form a sensible language, we must have that typed expansion of unexpanded expressions be consistent with typing of expanded expressions. Formally, this can be expressed as the following theorem.

Theorem 3.22 (Typed Expansion)**.** *If* $\Delta\ \Gamma \vdash_\Sigma \hat{e} \leadsto e : \tau$ *and* $\Delta \vdash \Sigma\ \mathsf{menv}$ *then* $\Delta\ \Gamma \vdash e : \tau$.

Proof. By rule induction over Rules (3.3).

Case (3.3a)**.** We have

(1)	$\hat{e} = x$	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 (3.3b)**.** We have

(1) $\hat{e} = \mathtt{ulam}\{\tau_1\}(x.\hat{e}')$ by assumption
(2) $e = \mathtt{elam}\{\tau_1\}(x.e')$ by assumption
(3) $\tau = \mathtt{parr}(\tau_1;\tau_2)$ by assumption
(4) $\Delta \vdash \tau_1\ \mathsf{type}$ by assumption
(5) $\Delta\ \Gamma, x : \tau_1 \vdash_\Sigma \hat{e}' \rightsquigarrow e' : \tau_2$ by assumption
(6) $\Delta \vdash \Sigma\ \mathsf{menv}$ by assumption
(7) $\Delta\ \Gamma, x : \tau_1 \vdash e' : \tau_2$ by IH on (5) and (6)
(8) $\Delta\ \Gamma \vdash \mathtt{elam}\{\tau_1\}(x.e') : \mathtt{parr}(\tau_1;\tau_2)$ by Rule (3.2b) on (4) and (7)

Case (3.3c)**.** We have

(1) $\hat{e} = \mathtt{uap}(\hat{e}_1;\hat{e}_2)$ by assumption
(2) $e = \mathtt{eap}(e_1;e_2)$ by assumption
(3) $\Delta\ \Gamma \vdash_\Sigma \hat{e}_1 \rightsquigarrow e_1 : \mathtt{parr}(\tau_1;\tau)$ by assumption
(4) $\Delta\ \Gamma \vdash_\Sigma \hat{e}_2 \rightsquigarrow e_2 : \tau_1$ by assumption
(5) $\Delta \vdash \Sigma\ \mathsf{menv}$ by assumption
(6) $\Delta\ \Gamma \vdash e_1 : \mathtt{parr}(\tau_1;\tau)$ by IH on (3) and (5)
(7) $\Delta\ \Gamma \vdash e_2 : \tau_1$ by IH on (4) and (5)
(8) $\Delta\ \Gamma \vdash \mathtt{eap}(e_1;e_2) : \tau$ by Rule (3.2c) on (6) and (7)

Case (3.3d) through (3.3k)**.** These cases follow analagously, i.e. we apply the IH to all typed expansion premises and then apply the corresponding typing rule.

Case (3.3l)**.** We have

(1) $\hat{e} = \mathtt{usyntax}\{\hat{e}_{\text{parse}}\}\{\tau'\}(a.\hat{e}')$ by assumption
(2) $\Delta \vdash \tau'\ \mathsf{type}$ by assumption
(3) $\emptyset\ \emptyset \vdash_\emptyset \hat{e}_{\text{parse}} \rightsquigarrow e_{\text{parse}} : \mathtt{parr}(\mathtt{Body};\mathtt{ParseResultExp})$ by assumption
(4) $\Delta\ \Gamma \vdash_{\Sigma, a \hookrightarrow \mathtt{syntax}(\tau';e_{\text{parse}})} \hat{e}' \rightsquigarrow e : \tau$ by assumption
(5) $\Delta \vdash \Sigma\ \mathsf{menv}$ by assumption
(6) $\emptyset \vdash \emptyset\ \mathsf{menv}$ by Rule (3.4a)
(7) $\emptyset\ \emptyset \vdash e_{\text{parse}} : \mathtt{parr}(\mathtt{Body};\mathtt{ParseResultExp})$ by IH on (3) and (6)
(8) $\Delta \vdash \Sigma, a \hookrightarrow \mathtt{syntax}(\tau'; e_{\text{parse}})\ \mathsf{menv}$ by Rule (3.4b) on (5), (2) and (7)
(9) $\Delta\ \Gamma \vdash e : \tau$ by IH on (4) and (8)

Case (3.3m)**.** We have

(1) $\hat{e} = \mathtt{utsmap}[b][a]$ by assumption

(2) $\Sigma = \Sigma', a \hookrightarrow \mathtt{syntax}(\tau; e_{\text{parse}})$ 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) $\emptyset\, \emptyset \vdash^{\Delta;\Gamma;\Sigma;b} \grave{e} \rightsquigarrow e : \tau$ by assumption

(7) $\Delta \vdash \Sigma\ \mathsf{menv}$ by assumption

(8) $\Delta\, \Gamma \vdash e : \tau$ by Theorem 3.23 on (6) and (7)

□

We need to define the following theorem about candidate expansion expression validation mutually with Theorem 3.22.

Theorem 3.23 (Candidate Expansion Expression Validation). *If* $\Delta\, \Gamma \vdash^{\Delta_S;\Gamma_S;\Sigma_S;b} \grave{e} \rightsquigarrow e : \tau$ *and* $\Delta_S \vdash \Sigma_S\ \mathsf{menv}$ *then* $\Delta \cup \Delta_S\ \Gamma \cup \Gamma_S \vdash e : \tau$.

Proof. By rule induction over Rules (3.6).

Case (3.6a). We have

(1) $\grave{e} = x$ by assumption

(2) $e = x$ by assumption

(3) $\Gamma = \Gamma', x : \tau$ by assumption

(4) $\Delta \cup \Delta_S\ \Gamma', x : \tau \vdash x : \tau$ by Rule (3.2a)

(5) $\Delta \cup \Delta_S\ \Gamma', x : \tau \cup \Gamma_S \vdash x : \tau$ by Lemma 3.1 over Γ_S to (4)

Case (3.6b). We have

(1) $\grave{e} = \mathtt{celam}\{\grave{\tau}_1\}(x.\grave{e}')$ by assumption

(2) $e = \mathtt{elam}\{\tau_1\}(x.e')$ by assumption

(3) $\tau = \mathtt{parr}(\tau_1;\tau_2)$ by assumption

(4) $\Delta \vdash^{\Delta_S;\Gamma_S;\Sigma_S;b} \grave{\tau}_1 \rightsquigarrow \tau_1\ \mathsf{type}$ by assumption

(5) $\Delta\, \Gamma, x : \tau_1 \vdash^{\Delta_S;\Gamma_S;\Sigma_S;b} \grave{e}' \rightsquigarrow e' : \tau_2$ by assumption

(6) $\Delta_S \vdash \Sigma_S\ \mathsf{menv}$ by assumption

(7) $\Delta \cup \Delta_S \vdash \tau_1\ \mathsf{type}$ by Lemma 3.26 on (4)

(8) $\Delta \cup \Delta_S\ \Gamma, x : \tau_1 \cup \Gamma_S \vdash e' : \tau_2$ by IH on (5) and (6)

(9) $\Delta \cup \Delta_S\ \Gamma \cup \Gamma_S, x : \tau_1 \vdash e' : \tau_2$ by exchange over Γ_S on (8)

(10) $\Delta \cup \Delta_S\ \Gamma \cup \Gamma_S \vdash \mathtt{elam}\{\tau_1\}(x.e') : \mathtt{parr}(\tau_1;\tau_2)$ by Rule (3.2b) on (7) and (9)

Case (3.6c). We have

(1) $\grave{e} = \mathtt{ceap}(\grave{e}_1;\grave{e}_2)$ by assumption

(2) $e = \texttt{eap}(e_1;e_2)$ by assumption
(3) $\Delta\ \Gamma \vdash^{\Delta_S;\Gamma_S;\Sigma_S;b} \grave{e}_1 \leadsto e_1 : \texttt{parr}(\tau_1;\tau)$ by assumption
(4) $\Delta\ \Gamma \vdash^{\Delta_S;\Gamma_S;\Sigma_S;b} \grave{e}_2 \leadsto e_2 : \tau_1$ by assumption
(5) $\Delta_S \vdash \Sigma_S\ \mathsf{menv}$ by assumption
(6) $\Delta \cup \Delta_S\ \Gamma \cup \Gamma_S \vdash e_1 : \texttt{parr}(\tau_1;\tau)$ by IH on (3) and (5)
(7) $\Delta \cup \Delta_S\ \Gamma \cup \Gamma_S \vdash e_2 : \tau_1$ by IH on (4) and (5)
(8) $\Delta \cup \Delta_S\ \Gamma \cup \Gamma_S \vdash \texttt{eap}(e_1;e_2) : \tau$ by Rule (3.2c) on (6) and (7)

Case (3.6d) through (3.6k)**.** These cases follow analagously, i.e. we apply the IH to all candidate expansion expression validation premises, Lemma 3.26 to all candidate expansion type validation premises, weakening and exchange as needed, and then apply the corresponding typing rule.

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) $\Delta_S\ \Gamma_S \vdash_{\Sigma_S} \hat{e} \leadsto e : \tau$ by assumption
(4) $\Delta_S \vdash \Sigma_S\ \mathsf{menv}$ by assumption
(5) $\Delta_S\ \Gamma_S \vdash e : \tau$ by Theorem 3.22 on (3) and (4)
(6) $\Delta \cup \Delta_S\ \Gamma \cup \Gamma_S \vdash e : \tau$ by Lemma 3.1 on (5)

□

The mutual induction used to prove Theorems 3.22 and 3.23 can be shown to be well-founded by showing that the following numeric metric on the judgements that we induct over is decreasing:

$$\begin{aligned} \|\Delta\ \Gamma \vdash_{\Sigma} \hat{e} \leadsto e : \tau\| &= \|\hat{e}\| \\ \|\Delta\ \Gamma \vdash^{\Delta_S;\Gamma_S;\Sigma_S;b} \grave{e} \leadsto e : \tau\| &= \|b\| \end{aligned}$$

Here, $\|b\|$ is the length of b and $\|\hat{e}\|$ is the sum of the lengths of the literal bodies in $\hat{e}$, i.e.

$$\begin{aligned} \|x\| &= 0 \\ \|\texttt{ulam}\{\tau\}(x.\hat{e})\| &= \|\hat{e}\| \\ \|\texttt{uap}(\hat{e}_1;\hat{e}_2)\| &= \|\hat{e}_1\| + \|\hat{e}_2\| \\ \|\texttt{utlam}(t.\hat{e})\| &= \|\hat{e}\| \\ \|\texttt{utap}\{\tau\}(\hat{e})\| &= \|\hat{e}\| \\ \|\texttt{ufold}\{t.\tau\}(\hat{e})\| &= \|\hat{e}\| \\ \|\texttt{uunfold}(\hat{e})\| &= \|\hat{e}\| \end{aligned}$$

$$
\begin{aligned}
\|\texttt{utpl}(\{i \hookrightarrow \hat{e}_i\}_{i\in L})\| &= \sum_{i\in L} \|\hat{e}_i\| \\
\|\texttt{upr}[\ell](\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{uin}[\ell]\{\{i \hookrightarrow \tau_i\}_{i\in L}\}(\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{ucase}(\hat{e};\{i \hookrightarrow x_i.\hat{e}_i\}_{i\in L})\| &= \|\hat{e}\| + \sum_{i\in L} \|\hat{e}_i\| \\
\|\texttt{usyntax}\{\hat{e}_{\text{parse}}\}\{\tau\}(a.\hat{e})\| &= \|\hat{e}\| \\
\|\texttt{utsmap}[b][a]\| &= \|b\|
\end{aligned}
$$

The only case in the proof of Theorem 3.22 that invokes Theorem 3.23 is Case (3.3m). There, we have that the metric remains stable:

$$\|\emptyset\,\emptyset \vdash^{\Delta;\Gamma;\Sigma;b} \grave{e} \leadsto e : \tau\| = \|\Delta\;\Gamma \vdash_{\Sigma} \texttt{utsmap}[b][a] \leadsto e : \tau\| = \|b\|$$

The only case in the proof of Theorem 3.23 that invokes Theorem 3.22 is Case (3.6l). There, we have that $\mathsf{parseUExp}(\mathsf{subseq}(b;m;n)) = \hat{e}$ and Theorem 3.22 is invoked on the judgement $\Delta_S\;\Gamma_S \vdash_{\Sigma_S} \hat{e} \leadsto e : \tau$. Because the metric is stable when passing from Theorem 3.22 to Theorem 3.23, we must have that it is strictly decreasing in the other direction:

$$\|\Delta_S\;\Gamma_S \vdash_{\Sigma_S} \hat{e} \leadsto e : \tau\| < \|\Delta\;\Gamma \vdash^{\Delta_S;\Gamma_S;\Sigma_S;b} \texttt{cesplicede}[m;n] \leadsto e : \tau\|$$

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

The second condition states that an unexpanded expression constructed by parsing a literal body 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 on and delimit each literal body.)

Condition 3.25 (Body Parsing)**.** *If* $\mathsf{parseUExp}(b) = \hat{e}$ *then* $\|\hat{e}\| < \|b\|$.

Combining Conditions 3.24 and 3.25, we have that $\|\hat{e}\| < \|b\|$ as needed.

□

Finally, the proof of Theorem 3.23 invokes the following lemma about candidate expansion type validation.

Lemma 3.26 (Candidate Expansion Type Validation)**.** *If* $\Delta \vdash^{\Delta_S;\Gamma_S;\Sigma_S;b} \grave{\tau} \leadsto \tau\ \mathsf{type}$ *then* $\Delta \cup \Delta_S \vdash \tau\ \mathsf{type}$.

Proof. By rule induction over Rules (3.5).

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

(1) $\Delta = \Delta', t\ \mathsf{type}$ by assumption
(2) $\grave{\tau} = t$ by assumption

(3) $\tau = t$ — by assumption

(4) $\Delta', t\ \mathsf{type} \vdash t\ \mathsf{type}$ — by Rule (3.1a)

(5) $\Delta', t\ \mathsf{type} \cup \Delta_S \vdash t\ \mathsf{type}$ — by Lemma 3.1 over Δ_S to (4).

Case (3.5b). We have

(1) $\check{\tau} = \mathtt{ceparr}(\check{\tau}_1; \check{\tau}_2)$ — by assumption

(2) $\tau = \mathtt{parr}(\tau_1; \tau_2)$ — by assumption

(3) $\Delta \vdash^{\Delta_S;\Gamma_S;\Sigma_S;b} \check{\tau}_1 \leadsto \tau_1\ \mathsf{type}$ — by assumption

(4) $\Delta \vdash^{\Delta_S;\Gamma_S;\Sigma_S;b} \check{\tau}_2 \leadsto \tau_2\ \mathsf{type}$ — by assumption

(5) $\Delta \cup \Delta_S \vdash \tau_1\ \mathsf{type}$ — by IH on (3)

(6) $\Delta \cup \Delta_S \vdash \tau_2\ \mathsf{type}$ — by IH on (4)

(7) $\Delta \cup \Delta_S \vdash \mathtt{parr}(\tau_1; \tau_2)\ \mathsf{type}$ — by Rule (3.1b) on (5) and (6)

Case (3.5c). We have

(1) $\check{\tau} = \mathtt{ceall}(t.\check{\tau}')$ — by assumption

(2) $\tau = \mathtt{all}(t.\tau')$ — by assumption

(3) $\Delta, t\ \mathsf{type} \vdash^{\Delta_S;\Gamma_S;\Sigma_S;b} \check{\tau}' \leadsto \tau'\ \mathsf{type}$ — by assumption

(4) $\Delta, t\ \mathsf{type} \cup \Delta_S \vdash \tau'\ \mathsf{type}$ — by IH on (3)

(5) $\Delta \cup \Delta_S, t\ \mathsf{type} \vdash \tau'\ \mathsf{type}$ — by exchange over Δ_S on (4)

(6) $\Delta \cup \Delta_S \vdash \mathtt{all}(t.\tau')\ \mathsf{type}$ — by Rule (3.1c) on (5)

Case (3.5d). We have

(1) $\check{\tau} = \mathtt{cerec}(t.\check{\tau}')$ — by assumption

(2) $\tau = \mathtt{rec}(t.\tau')$ — by assumption

(3) $\Delta, t\ \mathsf{type} \vdash^{\Delta_S;\Gamma_S;\Sigma_S;b} \check{\tau}' \leadsto \tau'\ \mathsf{type}$ — by assumption

(4) $\Delta, t\ \mathsf{type} \cup \Delta_S \vdash \tau'\ \mathsf{type}$ — by IH on (3)

(5) $\Delta \cup \Delta_S, t\ \mathsf{type} \vdash \tau'\ \mathsf{type}$ — by exchange over Δ_S on (4)

(6) $\Delta \cup \Delta_S \vdash \mathtt{rec}(t.\tau')\ \mathsf{type}$ — by Rule (3.1d) on (5)

Case (3.5e). We have

(1) $\check{\tau} = \mathtt{ceprod}(\{i \hookrightarrow \check{\tau}_i\}_{i \in L})$ — by assumption

(2) $\tau = \mathtt{prod}(\{i \hookrightarrow \tau_i\}_{i \in L})$ — by assumption

(3) $\{\Delta \vdash^{\Delta_S;\Gamma_S;\Sigma_S;b} \check{\tau}_i \leadsto \tau_i\ \mathsf{type}\}_{i \in L}$ — by assumption

(4) $\{\Delta \cup \Delta_S \vdash \tau_i\ \mathsf{type}\}_{i \in L}$ — by IH on $(3)_i$ for each $i \in L$

(5) $\Delta \cup \Delta_S \vdash \mathtt{prod}(\{i \hookrightarrow \tau_i\}_{i \in L})\ \mathsf{type}$ — by Rule (3.1e) on (4)

Case (3.5f). We have

(1) $\check{\tau} = \texttt{cesum}(\{i \hookrightarrow \check{\tau}_i\}_{i \in L})$ by assumption

(2) $\tau = \texttt{sum}(\{i \hookrightarrow \tau_i\}_{i \in L})$ by assumption

(3) $\{\Delta \vdash^{\Delta_S;\Gamma_S;\Sigma_S;b} \check{\tau}_i \rightsquigarrow \tau_i \; \mathsf{type}\}_{i \in L}$ by assumption

(4) $\{\Delta \cup \Delta_S \vdash \tau_i \; \mathsf{type}\}_{i \in L}$ by IH on $(3)_i$ for each $i \in L$

(5) $\Delta \cup \Delta_S \vdash \texttt{sum}(\{i \hookrightarrow \tau_i\}_{i \in L}) \; \mathsf{type}$ by Rule (3.1f) on (4)

Case (3.5g). We have

(1) $\check{\tau} = \texttt{cesplicedt}[m; n]$ by assumption

(2) $\mathsf{parseTyp}(\mathsf{subseq}(b; m; n)) = \tau$ by assumption

(3) $\Delta_S \vdash \tau \; \mathsf{type}$ by assumption

(4) $\Delta \cup \Delta_S \vdash \tau \; \mathsf{type}$ by Lemma 3.1 over Δ on (3) and exchange

□

Chapter 4

Unparameterized Pattern TSMs

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 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 decide whether match failure is possible (i.e. whether there are possible scrutinee values 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 result in a 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$_{\mathbf{U}}$ 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**, i.e. pattern TSMs that generate patterns that match values of a single specified type, like `Rx`. In Chapter 5, we will consider both expression and pattern TSMs defined over parameterized families of types.

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 unparameterized pattern TSMs in VerseML. In particular, we discuss a pattern TSM for patterns matching values of the recursive labeled sum type `Rx` just discussed. In the next section, we specify a reduced formal system based on miniVerse$_{\mathbf{U}}$ called miniVerse$_{\mathbf{UP}}$ that makes the intuitions developed in this section mathematically precise.

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 pattern TSM, `$rx`, as follows:

[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 an expression TSM might use that have no corresponding pattern form, e.g. lambda abstraction.

```
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 *)
}
```

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

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

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

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.

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$_{\mathbf{UP}}$

To make the intuitions developed in the previous section about pattern TSMs precise, we now introduce a reduced calculus with support for both unparameterized expression TSMs and unparameterized pattern TSMs called miniVerse$_{\mathbf{UP}}$.

4.2.1 Types, Expanded Expressions, Rules and Patterns

The *inner core* of miniVerse$_{\mathbf{UP}}$ 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$_{\mathbf{UP}}$ differs from that of miniVerse$_{\mathbf{U}}$ only in that the case analysis operator has been replaced by the pattern matching operator, so we will omit some definitions that are identical to those in Sec. 3.4. Our formulation of the semantics of pattern matching is adapted from the Harper's formulation in *Practical Foundations for Programming Languages, First Edition* [18].[2]

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

Sort			Abstract 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
			$\texttt{prod}(\{i \hookrightarrow \tau_i\}_{i \in L})$	$\langle\{i \hookrightarrow \tau_i\}_{i \in L}\rangle$	labeled product
			$\texttt{sum}(\{i \hookrightarrow \tau_i\}_{i \in L})$	$[\{i \hookrightarrow \tau_i\}_{i \in L}]$	labeled sum
EExp	e	$::=$	x	x	variable
			$\texttt{elam}\{\tau\}(x.e)$	$\lambda x{:}\tau.e$	abstraction
			$\texttt{eap}(e;e)$	$e(e)$	application
			$\texttt{etlam}(t.e)$	$\Lambda t.e$	type abstraction
			$\texttt{etap}\{\tau\}(e)$	$e[\tau]$	type application
			$\texttt{efold}\{t.\tau\}(e)$	$\texttt{fold}(e)$	fold
			$\texttt{eunfold}(e)$	$\texttt{unfold}(e)$	unfold
			$\texttt{etpl}(\{i \hookrightarrow e_i\}_{i \in L})$	$\langle\{i \hookrightarrow e_i\}_{i \in L}\rangle$	labeled tuple
			$\texttt{epr}[\ell](e)$	$e \cdot \ell$	projection
			$\texttt{ein}[\ell]\{\{i \hookrightarrow \tau_i\}_{i \in L}\}(e)$	$\ell \cdot e$	injection
			$\texttt{ematch}(e;\{r_i\}_{1 \leq i < n})$	$\texttt{match}\ e\ \{r_i\}_{1 \leq i < n}$	match
ERule	r	$::=$	$\texttt{erule}[n]\{p\}(\{x_i\}_{1 \leq i < n}.e)$	$p \Rightarrow e$	rule
EPat	p	$::=$	x	x	variable pattern
			$\texttt{ewildp}$	_	wildcard pattern
			$\texttt{etplp}(\{i \hookrightarrow p_i\}_{i \in L})$	$\langle\{i \hookrightarrow p_i\}_{i \in L}\rangle$	labeled tuple pattern
			$\texttt{einp}[\ell](p)$	$\ell \cdot p$	injection pattern

Figure 4.2: Syntax of types and expanded expressions, rules and patterns in miniVerse$_{\mathbf{UP}}$. We adopt the metatheoretic conventions established for our specification of miniVerse$_{\mathbf{U}}$ in Sec. 3.2 without restating them. We write $\{r_i\}_{1 \leq i < n}$ for sequences of $n > 0$ rules and $\{x_i\}_{1 \leq i < n}.e$ for expressions binding the sequence of $n > 0$ variables $\{x_i\}_{1 \leq i < n}$.

Statics of the Inner Core

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

Judgement Form	**Description**
$\Delta \vdash \tau\ \mathsf{type}$	τ is a well-formed type assuming Δ
$\Delta\ \Gamma \vdash e : \tau$	e has type τ assuming Δ and Γ
$\Delta\ \Gamma \vdash r : \tau \Rightarrow \tau'$	r takes values of type τ to values of type τ' assuming Δ and Γ
$\Omega \Vdash p : \tau$	p matches values of type τ and generates hypotheses Ω

The *type formation judgement*, $\Delta \vdash \tau\ \mathsf{type}$, 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 (3.2a) through (3.2j) extended with the following rule for match expressions:

$$\frac{\Delta\ \Gamma \vdash e : \tau \qquad \{\Delta\ \Gamma \vdash r_i : \tau \Rightarrow \tau'\}_{1 \le i < n}}{\Delta\ \Gamma \vdash \mathtt{ematch}(e; \{r_i\}_{1 \le i < n}) : \tau'} \tag{4.1}$$

The first premise of Rule (4.1) assigns a type, τ, to the scrutinee, e. The second premise then ensures that each rule r_i takes values of type τ to values of the type assigned to the match expression as a whole, τ'. This is expressed by the *rule typing judgement*, $\Delta\ \Gamma \vdash r : \tau \Rightarrow \tau'$, which is defined mutually with the typing rules by the following rule:

$$\frac{\Omega \Vdash p : \tau \qquad \mathrm{dom}(\Omega) = \{x_i\}_{1 \le i < n} \qquad \Delta\ \Gamma \cup \Omega \vdash e : \tau'}{\Delta\ \Gamma \vdash \mathtt{erule}[n]\{p\}(\{x_i\}_{1 \le i < n}.e) : \tau \Rightarrow \tau'} \tag{4.2}$$

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

1. The first premise invokes the *pattern typing judgement*, $\Omega \Vdash p : \tau$, to check that the pattern, p, matches values of type τ, and gather the typing hypotheses that the pattern generates in a *pattern typing context*, Ω. Pattern typing contexts, like typing contexts, Γ, consist of hypotheses $x : \tau$. We specify a distinct metavariable because the pattern typing judgement is a *linear hypothetical judgement*, i.e. it does not obey weakening. This is to ensure that the pattern typing context captures exactly those variables bound by a pattern. Algorithmically, one can think of the pattern typing context as an "output" of pattern typing.

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

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

$$\frac{}{\emptyset \Vdash \mathtt{ewildp} : \tau} \tag{4.3b}$$

$$\frac{\{\Omega_i \Vdash p_i : \tau_i\}_{i \in L} \qquad \cap_{i \in L} \mathrm{dom}(\Omega_i) = \emptyset}{\cup_{i \in L} \Omega_i \Vdash \mathtt{etplp}(\{i \hookrightarrow p_i\}_{i \in L}) : \mathtt{prod}(\{i \hookrightarrow \tau_i\}_{i \in L})} \tag{4.3c}$$

$$\frac{\Omega \Vdash p : \tau}{\Omega \Vdash \mathtt{einp}[\ell](p) : \mathtt{sum}(\{i \hookrightarrow \tau_i\}_{i \in L} \otimes \ell \hookrightarrow \tau)} \tag{4.3d}$$

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

Rule (4.3b) specifies that a wildcard pattern can also match values of any type, τ, but it generates no hypotheses.

Labeled tuple patterns, $\texttt{etplp}(\{i \hookrightarrow p_i\}_{i\in L})$, specify a subpattern p_i for each label $i \in L$. Rule (4.3c) specifies that a labeled tuple pattern of this form matches values of the labeled product type $\texttt{prod}(\{i \hookrightarrow \tau_i\}_{i\in L})$. Each subpattern p_i is checked against the corresponding type τ_i, generating hypotheses Ω_i. The rule then ensures that the subpatterns bind distinct variables by checking that the domains of the pattern typing contexts Ω_i are disjoint. The hypotheses generated in the conclusion of the rule is then the union of the hypotheses Ω_i.

Injection patterns, $\texttt{einp}[\ell](p)$, match values of labeled sum types of the form $\texttt{sum}(\{i \hookrightarrow \tau_i\}_{i\in L} \otimes \ell \hookrightarrow \tau)$, i.e. labeled sum types that specify a case for ℓ. Rule (4.3d) checks the subpattern p against the corresponding type τ, and passes along exactly those assumptions that p generates.

2. The second premise of Rule (4.2) ensures that the pattern has generated hypotheses for all of the the variables that the branch expression, e, binds. This is merely a matter of "metatheoretic bookkeeping". In the stylized form for a rule, $p \Rightarrow e$, the bound variables are, implicitly, exactly those found in the pattern.
3. The final premise of Rule (4.2) extends the typing context, Γ, with the hypotheses generated by pattern typing, Ω, and checks the branch expression, e, against the branch type, τ'.

Dynamics of miniVerse$_{\mathbf{UP}}$

The *dynamics of* miniVerse$_{\mathbf{UP}}$ is specified as a structural dynamics (a.k.a. structural operational semantics) by judgements of the following form:

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

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

Definition 4.1 (Iterated Transition)**.** *Iterated transition, $e \mapsto^* e'$, is the reflexive, transitive closure of the transition judgement.*

Definition 4.2 (Evaluation)**.** *$e \Downarrow e'$ iff $e \mapsto^* e'$ and e' val.*

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

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

Condition 4.3 (Canonical Forms)**.** *If $\vdash e : \tau$ and e val then:*

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

2. *If* $\tau = \texttt{all(}t.\tau'\texttt{)}$ *then* $e = \texttt{etlam(}t.e'\texttt{)}$ *and* $t\ \mathsf{type} \vdash e' : \tau'$.
3. *If* $\tau = \texttt{rec(}t.\tau'\texttt{)}$ *then* $e = \texttt{efold}\{t.\tau'\}\texttt{(}e'\texttt{)}$ *and* $\vdash e' : [\texttt{rec(}t.\tau'\texttt{)}/t]\tau'$.
4. *If* $\tau = \texttt{prod(}\{i \hookrightarrow \tau_i\}_{i \in L}\texttt{)}$ *then* $e = \texttt{etpl(}\{i \hookrightarrow e_i\}_{i \in L}\texttt{)}$ *and for each* $i \in L$ *we have that* $\vdash e_i : \tau_i$ *and* $e_i\ \mathsf{val}$.
5. *If* $\tau = \texttt{sum(}\{i \hookrightarrow \tau_i\}_{i \in L}\texttt{)}$ *then* $e = \texttt{ein[}\ell\texttt{]}\{\{i \hookrightarrow \tau_i\}_{i \in L \setminus \ell} \otimes \ell \hookrightarrow \tau'\}\texttt{(}e'\texttt{)}$ *and* $\ell \in L$ *and* $\vdash e' : \tau'$ *and* $e'\ \mathsf{val}$.

We also require that the statics and dynamics be coherent. This is expressed in the standard way by Preservation and Progress conditions (together, these constitute the Type Safety Condition).

Condition 4.4 (Preservation)**.** *If* $\vdash e : \tau$ *and* $e \mapsto e'$ *then* $\vdash e' : \tau$.

Condition 4.5 (Progress)**.** *If* $\vdash e : \tau$ *then either* $e\ \mathsf{val}$ *or there exists an* e' *such that* $e \mapsto e'$.

4.2.2 Unexpanded Expressions and Patterns

4.2.3 Pattern Macro Definition

4.2.4 Pattern Macro Application

4.2.5 Candidate Expansion Pattern Validation

4.2.6 Metatheory

Chapter 5

Parameterized TSMs

5.1 Parameterized TSMs By Example

5.1.1 Value Parameters By Example

5.1.2 Type Parameters By Example

5.1.3 Module Parameters By Example

5.2 miniVerse$_{\forall}$

5.2.1 Signatures, Types and Expanded Expressions

5.2.2 Parameter Application and Deferred Substitution

5.2.3 Macro Expansion and Validation

5.2.4 Metatheory

Chapter 6

Type-Specific Languages (TSLs)

With TSMs, library providers can control the expansion of arbitrary syntax that appears between delimiters, but clients must explicitly identify the TSM and provide the required type and module parameters at each use site. To further lower the syntactic cost of using TSMs, so that it compares to the syntactic cost of derived syntax built in primitively (e.g. list syntax), we will now discuss how VerseML allows library providers to define *type-specific languages* (TSLs) by associating a TSM directly with an abstract type or datatype. When the type system encounters a delimited form not prefixed by a TSM name, it applies the TSM associated with the type it is being analyzed against implicitly.

6.1 TSLs 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.

6.2 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 *)
```

6.3 miniVerse$_{\mathbf{TSL}}$

A formal specification of TSLs in a language that supports only non-parametric datatypes is available in a paper published in ECOOP 2014 [30]. We will add support for parameterized TSLs in the dissertation (see Sec. **??**).

Chapter 7

Discussion & Future Directions

7.1 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.

7.1.1 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.

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)
```

Parser Generators

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

7.1.2 Monadic Commands

7.2 Summary

TODO: Write summary

7.3 Future Directions

7.3.1 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 *)
```

7.3.2 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. In VerseML, expansion validation occurs assuming the type equalities that are known at the definition site, to avoid this problem.

7.3.3 TSMs and TSLs In Candidate Expansions

TODO: write this

7.3.4 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`, shown in Sec. **??**. We leave further discussion of such a facility and of parameterized TPSMs also as remaining work (see Sec. **??**).

7.3.5 Integration Into Other Languages

We conjecture that the constructs we describe could be integrated into dependently typed functional languages, e.g. Coq, but leave the technical developments necessary for doing so as future work.

Some of the constructs in Chapter 3, Chapter 5 and Chapter 6 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 6 are not relevant to such languages.

7.3.6 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 [26]) 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

7.3.7 Improved Error Reporting

7.3.8 Controlled Binding

7.3.9 Type-Aware Splicing

7.3.10 Integration With Code Editors

7.3.11 Resugaring

TODO: Cite recent work at PLDI (?) and ICFP from Brown

7.3.12 Non-Textual Display Forms

TODO: Talk about active code completion work and future ideas

LaTeX Source Code and Updates

The LaTeX sources for this document can be found at the following URL:

https://github.com/cyrus-/thesis

The latest version of this document can be downloaded from the following URL:

https://github.com/cyrus-/thesis/raw/master/omar-thesis.pdf

Any errors or omissions can be reported to the author by email at the following address:

comar@cs.cmu.edu

The author will also review and accept pull requests on GitHub.

Bibliography

TODO (Later): List conference abbreviations.

TODO (Later): Remove extraneous nonsense from entries.

[1] [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

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

[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

[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] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL http://coq.inria.fr. Version 8.0. 2.3.2, 7.3.6

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

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

[29] 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

[30] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely composable type-specific languages. In *ECOOP '14*, 2014. 3.1, 6.3

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

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

[33] 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

[34] 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

[35] 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

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

[37] 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

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

[39] 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

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

[41] Martin P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15

(4):147–161, 1994. 1.1.1

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