Modularly Programmable Syntax

Cyrus Omar

November 3, 2015

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jonathan Aldrich, Chair
TODO: confirm rest of committee

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

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, while nearly all major functional languages build in derived forms for lists, introducing derived forms for other library constructs, e.g. for HTML trees or regular expressions, requires forming new syntactic dialects of these languages. Unfortunately, programmers have no way to modularly combine such 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 syntactic dialects by giving library providers the ability to programmatically control syntactic expansion in a safe, hygienic and modular manner.

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) [32]

1.1 Motivation

Functional programming languages come in many sizes. The smallest functional languages – often referred to as "lambda calculi" – allow researchers and language designers to study the mathematical properties of language primitives of interest in isolation. These "small-scale" languages inform the design of "full-scale" languages, which combine several such primitives and also define various derived syntactic forms (colloquially, "syntactic sugar") that decrease the syntactic cost of selected idioms. For example, Standard ML (SML) [16, 25], OCaml [23] and Haskell [21] build in derived forms that decrease the syntactic cost of working with lists. For example, 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 [7]) that builds in derived forms for SQL queries, HTML elements and other datatypes used in the domain of web programming [8]. We will consider a large number of other examples of syntactic dialects in Sec. 2.1. Tools like Camlp4 [23], Sugar* [10, 11] and Racket's preprocessor [12], which we will discuss in Sec. 2.2, 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 [39]. 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 syntactic dialect featuring HTML syntax and one featuring regular expression syntax, but it is not always straightforward to construct a dialect featuring both. Such a dialect might be desirable for constructing, for example, a web-based bioinformatics tool.

In some cases, coming up with a combined dialect is difficult because the dialects are specified using different formalisms. In other cases, a common formalism has been used, but it does not operationalize the notion of dialect combination (e.g. Racket's preprocessor [12]). But even if we restrict our interest to dialects specified using a common formalism that does operationalize some notion of dialect combination, 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 [33], but it has limited expressive power, as we will discuss in Sec. 2.2.2.

1.1.2 Large Languages Considered Harmful

Dialects 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 discuss in Sec. 2.1, there are simply too many potentially useful such primitives, and many of these are only relevant 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 all these reasons, language designers are justifiably reticent to add new primitives to major programming languages.

1.1.3 Toward More General Primitives

If, as we've argued, language designers should strive both to avoid dialect formation and to keep general-purpose languages small, stable and free of *ad hoc* primitives, this leaves two possible paths forward. One, exemplified (arguably) by SML, is to simply eschew "niche" embellishments and settle on the existing design, 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 [40]. 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 the work being proposed is to take further steps down the second path just described by introducing primitive language constructs that reduce the need for syntactic dialects and *ad hoc* derived syntactic forms. In particular, we plan to introduce the following primitives:

1. **Typed syntax macros** (TSMs) give library providers programmatic control at compile-time over the parsing and expansion of literal forms. We will introduce TSMs first in the context of a simple language of expressions and types in Chapter 3, then add support for pattern matching in Chapter 4 and parameterized types and modules in Chapter 5.
2. **Type-specific languages** (TSLs), described in Chapter 6, further reduce syntactic cost by allowing library providers to associate a TSM with a type declaration and then rely on a local type inference scheme to invoke that TSM and apply its parameters implicitly.

As vehicles for this work, we will specify a small-scale typed lambda calculus in each of the chapters just mentioned. For the sake of examples, we will also describe (but not formally specify) a "full-scale" functional language called VerseML.[1] VerseML is, as its name suggests, a dialect of ML. It diverges from other dialects of ML that have a similar underlying type structure, like Standard ML and OCaml, in that it uses a local type inference scheme [31] (like, for example, Scala [26]) for reasons that have to do with the mechanisms described in Chapter 6. The reason we will not follow Standard ML [25] in giving a complete formal specification of VerseML here is both to emphasize that the primitives we introduce can be considered for inclusion in a variety of language designs (not exclusively dialects of ML), and to avoid distracting the reader with specifications for primitives that are already well-understood in the literature and that are orthogonal to those we introduce here.

The primitives we introduce perform *static code generation* (also sometimes called *static* or *compile-time metaprogramming*), meaning that the relevant rules in the static semantics of the language call for the evaluation of *static functions* that generate term encodings. Static functions are functions written in a restricted subset of the language called the *static language* (we will discuss the design space of restrictions in Sec. 3.3.4). 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

[1]We distinguish VerseML from Wyvern, which is the language referred to in prior publications about some of the work that we will describe, because Wyvern is a group effort evolving independently in some important ways.

system (but did not take an approach rooted in the study of type systems) [38].

The main challenge in the design of the primitives we introduce will come in ensuring that they are metatheoretically well-behaved. If we are not careful, many of the problems that arise when combining language dialects, discussed earlier, could simply shift into the semantics of these primitives.[2] Our main technical contributions will be to rigorously address these problems in a principled manner. In particular, we will maintain:

- a *type discipline*, meaning that the language is type safe, and that programmers reading a well-typed expression can determine its type without examining its expansion;
- a *hygiene discipline* (a.k.a. a *binding discipline*), meaning that the expansion logic does not make any assumptions about the names of variables in the context surrounding the expansion, nor does it introduce "hidden bindings" into subexpressions; and
- *modular reasoning principles*, meaning that library providers will have the ability to reason about the constructs that they have defined in isolation, and clients will be able to use them safely in any combination, without the possibility of conflict.[3]

We will make these notions more precise as we continue.

Thesis Statement

In summary, we will defend the following thesis statement:

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

1.3 Disclaimers

Before we continue, it may be useful 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 the language in this 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 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 [15], which also have some potential for "abuse" or "overuse").

[2]This is why languages like VerseML are often called "extensible languages", though this is somewhat of a misnomer. The defining characteristic of an extensible language is that it *doesn't* need to be extended in situations where other languages would need to be extended. We will avoid this somewhat confusing terminology.

[3]This is not quite true – simple naming conflicts can arise. We will tacitly assume that they are being avoided extrinsically, e.g. by using a URI-based naming scheme as in the Java ecosystem.

Chapter 2

Background

2.1 Motivating Examples

To further motivate our work, we will now provide a number of examples of derived syntactic forms that decrease the syntactic cost of working with various data structures. We cover the first two examples – regular expressions and lists – in substantial detail. We will refer back to these examples throughout this work. We then more concisely survey a number of other examples, grouped into categories, to establish the broad applicability of our contributions.

2.1.1 Regular Expressions

Let us take the perspective of a regular expression library provider (we assume the reader has some familiarity with regular expressions [37]). 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 [17]. In VerseML, a labeled recursive sum type can be defined like this:

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

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

The abstract syntax of regexes is too verbose to be practical in all but the most trivial examples, so the POSIX standard specifies a more concise concrete syntax [2]. A number of programming languages support derived syntax for regular expressions based on this standard, e.g. Perl [9]. Let us consider a hypothetical dialect of ML called ML+Rx (perhaps constructed using a tool like Camlp4, discussed in Sec. 2.2.2) that similarly builds in derived forms for regexes (we will compare VerseML to ML+Rx in later chapters). ML+Rx supports *regex literals*, e.g.

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

desugars to:

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

ML+Rx also supports *spliced subexpressions* in regex literals. For example, the function `example_rx` shown below constructs a regex by splicing in a string, `name`, and another regex, `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 syntax that supports spliced subexpressions.

The prefix `@` indicates that `name` should be spliced in as a string, and the prefix `%` 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))
```

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 on a value of type `Rx` using derived pattern syntax. For example:

```
fun read_example_rx(r : Rx) =>
  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) =>
  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, $\textbf{seq}(\textbf{empty}, r)$ 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` where the type of patterns, `t`, is held abstract:

```
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.

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. 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

2.1.2 Lists

TODO: write this (Spring 2016)

2.1.3 Sets, Maps, Vectors and Other Containers

TODO: write this (Spring 2016)

2.1.4 HTML and Other Web Languages

TODO: write this; cite Ur/Web (Spring 2016)

2.1.5 Dates, URLs and Other Standardized Formats

TODO: write this (Spring 2016)

2.1.6 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 different database technologies, e.g. SQL for relational databases. TODO: finish this (Spring 2016)

2.1.7 Monadic Commands

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

2.1.8 Quasiquotation

TODO: write this (Spring 2016)

2.1.9 Grammars

TODO: write this (Spring 2016)

2.1.10 Mathematical and Scientific Notations

SMILES: Chemical Notation

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

TeX Mathematical Formula Notation

TODO: write this (Spring 2016)

2.2 Existing Approaches

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

2.2.1 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 esca` 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 [27]. 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.

[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`.

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 [3]. These are, of course, a consequence of the programmer making a mistake in an effort to decrease syntactic cost, but proving that mistakes like this have not been made involves reasoning about complex run-time data flows, so it is once again notoriously difficult to automate. If our language supported derived syntax for patterns, this kind of mistake would be substantially less common (because `example_shorter` has lower syntactic cost than `example_bad`).

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

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

   Though malformed patterns can sometimes be discovered dynamically via testing, empirical data gathered from large open source projects suggests that there remain many malformed regular expression patterns that are not detected by a project's test suite "in the wild" [35].

   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 [20]) also does not suffice.

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

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

5. Dynamic string parsing also necessarily incurs dynamic cost. Regular expression patterns are common when processing large datasets, so it is easy to inadvertently incur this cost repeatedly. For example, consider mapping over a list of strings:

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

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

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

The problems above are not unique to regular expression patterns. Whenever a library encourages the use of dynamic string parsing to address the issue of syntactic cost (which is, fundamentally, not a dynamic issue), these problems arise. This fact has motivated much research on reducing the need for dynamic string parsing [5]. Existing alternatives can be broadly classified as being based on either *direct syntax extension* or *static term rewriting*. We describe these next, in Secs. 2.2.2 and 2.2.3 respectively.

2.2.2 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 [24]. A formal account of such a system has been developed by Griffin [14]. 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* [10] and Camlp4 [23] (amongst many others). Other systems give library providers direct programmatic access to the parse stream, like Common Lisp's *reader macros* [36] (which are distinct from its term-rewriting macros, described in Sec. 2.2.3 below) and Racket's preprocessor [12]. All of these would allow us to add pattern syntax into our language's grammar, perhaps following Unix conventions and supporting splicing syntax as described above:

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

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

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

In response to this problem, Schwerdfeger and Van Wyk developed a modular analysis that accepts only context-free grammar extensions that begin with an identifying starting token and obey certain constraints on the follow sets of base language's non-terminals [33]. 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

2.2.3 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 [18] 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 [22]. 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 [13, 19] and integrated into languages, e.g. MacroML [13] and Scala [6].

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 [1]). 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 [4] and Scala [34], 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.

Chapter 3

Unparameterized Expression TSMs

We now introduce a new primitive – the **typed syntax macro** (TSM) – that combines much of the syntactic flexibility of syntax extensions with the reasoning guarantees of typed macros. This chapter focuses on TSMs that generate expressions of a single specified type (*unparameterized expression TSMs*). We will add pattern matching in Chapter 4 and parameterized families of types in Chapter 5.

We begin in Sec. 3.1 by describing expression TSMs in VerseML by example. In particular, we will show a TSM for introducing values of the type `Rx` defined in Figure 2.1. We will then discuss examples of TSMs that are useful for defining other TSMs in Sec. 3.2. Finally, we will formally specify unparameterized expression TSMs with a lambda calculus, $\mathcal{L}^{\text{TSM}}$, in Sec. 3.3.

3.1 Expression TSMs By Example

3.1.1 Usage

Consider the following VerseML expression:

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

Here, we apply a TSM, identified as `$rx`, to a *literal form*, `/A|T|G|C/`. The TSM statically parses the *body* of the provided literal form, i.e. the characters in blue, to generate a *candidate expansion*. The language then *validates* the candidate expansion to maintain a type and hygiene discipline, in a manner that we will describe in Sec. 3.1.4, before producing the *final expansion*. The final expansion of the expression above, written concretely, is:

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

The constructors in the expansion above are those of the type `Rx` defined in Fig. 2.1.

A number of literal forms, shown in Figure 3.1, are available in VerseML's concrete syntax. Though certain TSMs may by convention call for the use of particular literal forms, any literal form can be used with any TSM, e.g. we could have written `$rx ‘A|T|G|C‘` above (this would actually be convenient if we wanted to write a regex containing forward slashes but not backticks).

```
'body cannot contain an apostrophe'
`body cannot contain a backtick`
[body cannot contain unmatched square brackets]
{body cannot contain an unmatched curly brace}
/body cannot contain a forward slash/
\body cannot contain a backslash\
42 (* numeric forms *)
42px (* numeric forms with suffixes *)
```

Figure 3.1: Literal forms available for use with TSMs in VerseML's concrete syntax. The characters in blue are the bodies of each form. In this figure, each line describes how the body is constrained by the form shown on that line. The Wyvern language specifies additional forms, including whitespace-delimited forms [28] and multipart forms [29], but for simplicity we leave these out of VerseML.

3.1.2 Definition

The definition of the TSM `$rx` shown in use above has the following form:

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

This TSM definition first identifies the TSM as `$rx`, then specifies a *type annotation*, `at Rx`, and a *parse function*, within curly braces.

VerseML enforces the convention that all TSM identifiers must be prefixed with a dollar sign (though other language designs could choose not to do so).

The parse function is a *static function* that generates an encoding of the candidate expansion, or indicates an error if a candidate expansion can't be generated (i.e. there is a parse error). Static functions are functions written in a static subset of the language, called the *static language* (SL). We will return to the design space around the static language in Sec. 3.3.4. The parse function has type `Body -> ParseResult`. These types are defined in the VerseML *prelude*, which is a set of definitions available ambiently. The input type, `Body`, gives the parse function access to the body of the provided literal form, which can be represented as a string:

```
type Body = string
```

The output type, `ParseResult`, distinguishes between successful parses and parse errors:

```
type ParseResult = Success of CandidateExp
                 | ParseError of {errorMsg : string, errorLoc : IndexRange}
```

Successful parses, constructed by `Success`, generate candidate expansions, which are encoded as values of type `CandidateExp`:

```
type CandidateExp = Var of var_t | ... | Spliced of IndexRange
```

The elided constructors above simply encode the abstract syntax of the VerseML expression language (as in the SML visible compiler [1]). We will discuss the final constructor, `Spliced`, in Sec. 3.1.3. We will show a complete encoding when we describe our reduced formal system $\mathcal{L}^{\text{TSM}}$ in Sec. 3.3. We define `CandidateExp` as a recursive labeled sum type for simplicity here, but alternative encodings of abstract syntax, e.g. based on abstract binding trees [17], could also have been chosen with only minor modification to the semantics.

If a parse error is detected, the parse function returns a value constructed by `ParseError`. The TSM provider must provide an error message (as a `string`) and indicate the location of the error relative to the body of the literal form as an index range, of type `IndexRange`:

```
type IndexRange = {startIndex : nat, endIndex : nat} (* inclusive *)
```

The error message and error location can be used by VerseML compilers when reporting errors to the programmer.

3.1.3 Splicing

To support spliced subexpressions, like we described in Sec. 2.1.1, 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, rewritten 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*. When the parse function determines that a subsequence of the body should be treated as a spliced subexpression (here, by recognizing the characters `@` or `%` followed by a variable or parenthesized expression), it can mark this subsequence as such and use it within the expansion it generates using the `Spliced` constructor of the `CandidateExp` type shown above. Notice that spliced subexpressions are referred to indirectly by their position within the literal body (i.e. with a value of type `IndexRange`, shown above) to prevent 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).

The candidate expansion generated by `$rx` for the body of `example_rx_tsm`, if written in a hypothetical concrete syntax for candidate expansions where 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 Validation

The language *validates* candidate expansions to ensure that they are *well-typed* and *hygienic*.

Typing

The type annotation specified by the TSM, i.e. the type `Rx` in the example above, determines the type that the candidate expansion is checked against. This maintains a type discipline: if a programmer encounters a TSM application when reading a well-typed program, they need only look up the TSM's type annotation to determine the type of that expression. They do not need to examine the expansion directly.

Hygiene

Hygiene, a.k.a. a binding discipline, is achieved by maintaining the following properties:

- **Context Independence**: A TSM cannot make any assumptions about the typing context at the use site. This is achieved by giving access to the use site typing context only to spliced subexpressions (which the client of the TSM writes, rather than the TSM provider).

  An example of a "bad" TSM that assumes that a particular variable is available at the use site is the following:

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

  The candidate expansion this TSM generates is well-typed only when there is a binding `x : Rx` at the use site. Therefore, it is deemed invalid.

  In the example in Sec. 3.1.1, the expansion used constructors associated with the `Rx` type, e.g. `Seq` and `Str`. This might appear to violate context independence. However, this is not the case because in VerseML, constructor labels are distinguished from variables. Syntactically, they must begin with a capital letter (as in Haskell). Different type definitions can use the same constructor label without conflict because the type the term is being checked against – here `Rx`, due to the type ascription on `$rx` – determines which type of value is to be constructed.

- **Context Invariance**: Spliced subexpressions have access to only those variables that were bound at the use site. The TSM cannot introduce new bindings into spliced subexpressions. For example, consider the following candidate expansion (written concretely as above):

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

  The variable `x` would not be available to the indicated spliced subexpression (nor would it shadow any bindings of `x` at the use site).

  This does limit the ability of library providers to define derived forms for abstractions that involve novel syntax for binding. For example, Haskell's derived forms for monadic commands involve binding the result of a command to a variable. In VerseML, programmers must use the built-in function form to pass data through variables. We will return to this example when we discuss parameterized TSMs in Chapter 5.

These properties suffice to ensure that programmers and tools can freely rename a variable without changing the meaning of the program. The only information that is necessary to perform such a *rename refactoring* is the location of spliced subexpressions within all the literal forms in scope of the variable; the expansions need not otherwise be examined. It would be straightforward to develop a tool and/or editor plugin to indicate the locations of spliced subexpressions to the user, just as we do in this document (by coloring spliced subexpressions black). We discuss tool support as future work in Sec. 7.2.4.

TODO: does above paragraph need to expanded?

Final Expansion

After checking that the candidate expansion is valid, the semantics replaces the references to spliced subexpressions with the subexpressions themselves, producing the final expansion. For example, the final expansion of the body of `example_rx_tsm` is:

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

3.2 TSMs For Defining TSMs

3.2.1 Quasiquotation

TSMs can be used when defining parse functions. For example, rather than construct values of type `MarkedExp` explicitly, the prelude includes a TSM that provides quasiquotation syntax like that described in Sec. 2.1.8:

```
syntax $qq at MarkedExp {
  static fn(body) => (* quasiquotation parser here *)
}
```

For example, the following TSM application:

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

is more concise than its expansion:

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

Anti-quotation is simply splicing in another value of type `MarkedExp`, here indicated by prefix `%`:

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

The expansion of this term is:

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

3.2.2 Parser Generators

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

3.3 Minimal Formalization

To give a formal account of unparameterized expression TSMs, we will now introduce a typed lambda calculus, $\mathcal{L}^{\text{TSM}}$. The syntax of $\mathcal{L}^{\text{TSM}}$ is shown in Figure 3.2.

3.3.1 Types and Expanded Expressions

Types, τ, and *expanded expressions*, e, form a standard typed lambda calculus supporting partial functions, quantification over types, recursive types and for simplicity, a single base type, 1. The reader can consult any standard text covering typed programming languages for the necessary background (e.g. *TAPL* [30] or *PFPL* [17]). We will reproduce a more detailed account of the

variables	**type variables**	**TSM variables**	**bodies**
x	t	m	b

types
$\tau ::= t \mid \tau \rightharpoonup \tau \mid \forall t.\tau \mid \mu t.\tau \mid 1$

marked types
$\dot{\tau} ::= t \mid \dot{\tau} \rightharpoonup \dot{\tau} \mid \forall t.\dot{\tau} \mid \mu t.\dot{\tau} \mid 1 \mid \mathsf{spliced}(\tau)$

TSM expressions
$\eta ::= m \mid \mathsf{syntax}\ @\ \tau\ \{\hat{e}\}$

unexpanded expressions
$\hat{e} ::= x \mid \lambda x{:}\tau.\hat{e} \mid \hat{e}(\hat{e}) \mid \Lambda t.\hat{e} \mid \hat{e}[\tau] \mid \mathsf{fold}[t.\tau](\hat{e}) \mid \mathsf{unfold}(\hat{e}) \mid () \mid \mathsf{let\ syntax}\ m = \eta\ \mathsf{in}\ \hat{e} \mid \eta\ /b/$

marked expressions
$\dot{e} ::= x \mid \lambda x{:}\dot{\tau}.\dot{e} \mid \dot{e}(\dot{e}) \mid \Lambda t.\dot{e} \mid \dot{e}[\dot{\tau}] \mid \mathsf{fold}[t.\dot{\tau}](\dot{e}) \mid \mathsf{unfold}(\dot{e}) \mid () \mid \mathsf{spliced}(e)$

expanded expressions
$e ::= x \mid \lambda x{:}\tau.e \mid e(e) \mid \Lambda t.e \mid e[\tau] \mid \mathsf{fold}[t.\tau](e) \mid \mathsf{unfold}(e) \mid ()$

type formation contexts	**typing contexts**
$\Delta ::= \emptyset \mid \Delta, t$	$\Gamma ::= \emptyset \mid \Gamma, x : \tau$

TODO: add sum and product types
TODO: add nat

Figure 3.2: Syntax of $\mathcal{L}^{\text{TSM}}$

TODO: write this down

Figure 3.3: Static semantics of expanded expressions in $\mathcal{L}^{\text{TSM}}$

TODO: write this down

Figure 3.4: Dynamic semantics of $\mathcal{L}^{\text{TSM}}$

TODO: write this down

Figure 3.5: Typed Expansion

```
(* we leave var and tvar abstract in the metatheory; see text *)
type var
type tvar
type Nat = Z | S of Nat
type Ty = TVar of tvar | Parr of Ty * Ty | Forall of tvar * Ty
  | Rec of tvar * Ty | Unit | Prod of Ty * Ty | Sum of Ty * Ty
  | SplicedTy of Nat * Nat
type Exp = Var of var | Abs of var * Ty * Exp | App of Exp * Exp
  | TyAbs of tvar * Exp | TyApp of Exp * Ty
  | Fold of tvar * Ty * Exp | Unfold of Exp
  | Triv | Pair of Exp * Exp | Fst of Exp | Snd of Exp
  | InL of Ty * Exp | InR of Ty * Exp | Case of Exp * var*Exp * var*Exp
  | SplicedExp of Nat * Nat
```

TODO: use math font TODO: reference tihs figure
TODO: ack. that there are other possible encodings that might be useful in practice, e.g. ABTs

Figure 3.6: Definition of types for expansion encodings in $\mathcal{L}^{\text{TSM}}$. For clarity, we use OCaml-style type definitions with explicit constructor labels here and in the remainder of this section. The mapping from these definitions to the unlabeled recursive sum types found in $\mathcal{L}^{\text{TSM}}$ is straightforward (e.g. `Nat` corresponds to $\mu t.1 + t$) and thus omitted for concision.

semantics of the language in the dissertation, but for our present purposes, it suffices to recall the relevant judgement forms. The static semantics of expanded expressions can be specified by judgements of the following form:

Judgement Form	**Description**
$\Delta \vdash \tau$ type	τ is a valid type under type formation context Δ
$\Delta \vdash \Gamma$ ctx	Γ is a valid typing context under Δ
$\Delta\ \Gamma \vdash e : \tau$	e has type τ under Δ and Γ

The dynamic semantics can be specified by judgements of the following form:

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

We will write $e \mapsto^* e'$ for the reflexive, transitive closure of the transition relation and $e \Downarrow e'$ iff $e \mapsto^* e'$ and e' val.

TODO: write this down

Figure 3.7: Expansion Decoding

TODO: write this down

Figure 3.8: Expansion Validation

3.3.2 Macro Expansion and Validation

Programs ultimately evaluate as expanded expressions, but programmers write programs as *unexpanded expressions*, $\hat{e}$. Unexpanded expressions are typechecked and expanded simultaneously according to the rules defining the *typed expansion judgement*:

Judgement Form	**Description**
$\Delta\,\Gamma \vdash \hat{e} \rightsquigarrow e : \tau$	$\hat{e}$ expands to e at type τ under Δ and Γ

Every form in the syntax of e has a corresponding form in the syntax of $\hat{e}$ (cf. Figure 3.2). For each rule in the static semantics of e, there is a corresponding typed expansion rule where the unexpanded and expanded forms correspond. For example, the rules for variables, functions and function application are shown below (the remaining such rules are analagous, but we omit them here for concision):

$$\textsc{T-U-var}\quad \frac{}{\Delta\,\Gamma, x : \tau \vdash x \rightsquigarrow x : \tau}$$

$$\textsc{T-U-abs}\quad \frac{\Delta \vdash \tau_1\ \texttt{type} \qquad \Delta\,\Gamma, x : \tau \vdash \hat{e} \rightsquigarrow e : \tau'}{\Delta\,\Gamma \vdash \lambda x{:}\tau.\hat{e} \rightsquigarrow \lambda x{:}\tau.e : \tau \rightharpoonup \tau'}$$

$$\textsc{T-U-ap}\quad \frac{\Delta\,\Gamma \vdash \hat{e}_1 \rightsquigarrow e_1 : \tau \rightharpoonup \tau' \qquad \Delta\,\Gamma \vdash \hat{e}_2 \rightsquigarrow e_2 : \tau}{\Delta\,\Gamma \vdash \hat{e}_1(\hat{e}_2) \rightsquigarrow e_1(e_2) : \tau'}$$

There are two forms in the syntax of $\hat{e}$ that have no corresponding form in the syntax of e. The first allows the programmer to bind a *TSM variable*, m, to a *TSM expression*, η. TSM expressions are either TSM variables or *TSM definitions*. Substitution for TSM variables is performed statically, so we only need a rule for the case where the TSM variable is being bound to a TSM definition:

$$\textsc{T-U-TSM-let}\quad \frac{\Delta \vdash \mathsf{syntax}\ @\ \tau\ \{\hat{e}_{\text{parse}}\}\ \texttt{tsm} \qquad \Delta\,\Gamma \vdash [\mathsf{syntax}\ @\ \tau\ \{\hat{e}_{\text{parse}}\}/m]\hat{e} \rightsquigarrow e : \tau'}{\Delta\,\Gamma \vdash \mathsf{let\ syntax}\ m = \mathsf{syntax}\ @\ \tau\ \{\hat{e}_{\text{parse}}\}\ \mathsf{in}\ \hat{e} \rightsquigarrow e : \tau'}$$

The first premise checks that the TSM definition is valid. It is defined by the following rule:

$$\textsc{TSM-ok}\quad \frac{\Delta \vdash \tau\ \texttt{type} \qquad \emptyset\,\emptyset \vdash \hat{e}_{\text{parse}} \rightsquigarrow e_{\text{parse}} : \mathsf{Body} \rightharpoonup \mathsf{Exp}}{\Delta \vdash \mathsf{syntax}\ @\ \tau\ \{\hat{e}_{\text{parse}}\}\ \texttt{tsm}}$$

The first premise of (TSM-OK) checks that the type is valid. The second premise typechecks and expands the parse function, which must be closed. We discuss the abbreviated types Body and Exp below.

The second form in the syntax of $\hat{e}$ that has no corresponding form in the syntax of e is the form for TSM application to a delimited body, $\eta\,/b/$. Again because substitution for TSM variables is performed statically, we only need a rule for the case where η is a TSM definition:

$$\frac{\begin{array}{c}\Delta \vdash \tau\ \mathtt{type} \qquad \emptyset\,\emptyset \vdash \hat{e}_{\text{parse}} \rightsquigarrow e_{\text{parse}} : \mathsf{Body} \rightharpoonup \mathsf{Exp} \\ b \downarrow e_{\text{body}} : \mathsf{Body} \qquad e_{\text{parse}}(e_{\text{body}}) \Downarrow e_{\text{exp}} \qquad e_{\text{exp}} : \mathsf{Exp} \uparrow \dot{e}_{\text{exp}} \\ \Delta\ \Gamma; \emptyset\,\emptyset \vdash \dot{e}_{\text{exp}} \rightsquigarrow e : \tau\end{array}}{\Delta\ \Gamma \vdash \mathsf{syntax}\ @\ \tau\ \{\hat{e}_{\text{parse}}\}\ /b/ \rightsquigarrow e : \tau}\ \textsc{T-U-TSM-ap}$$

The premises can be understood as follows, in order:

1. The first premise checks that the type specified by the TSM is valid.
2. The second premise typechecks and expands the parse function.
3. The third premise encodes the body, b, as a term, e_{body} of type Body (we will give more details on how bodies are encoded in the dissertation, but omit the definition of Body here for concision).
4. The fourth premise applies the expanded parse function to the encoding of the body to produce an encoding of the expansion, e_{exp}, of type Exp.
5. Values of type Exp map onto *marked expressions*, $\dot{e}$. Per Figure 3.2, marked expressions can contain variables, type variables and marked types, $\dot{\tau}$, so there is also a mapping from values of types abbreviated Var, TVar and Type onto variables, type variables, and marked types, respectively. We also omit the full definitions of these types for concision (cf. the SML/NJ Visible Compiler library [1] for an example of such an encoding of the abstract syntax of a language). These mappings are specified by the *expansion decoding judgements*:

Judgement Form	**Description**
$e : \mathsf{Var} \uparrow x$	e decodes to variable x.
$e : \mathsf{TVar} \uparrow t$	e decodes to type variable t.
$e : \mathsf{Type} \uparrow \dot{\tau}$	e decodes to marked type $\dot{\tau}$.
$e : \mathsf{Exp} \uparrow \dot{e}$	e decodes to marked expression $\dot{e}$.

   The fifth premise decodes e_{exp} to produce the *marked expansion*, $\dot{e}_{\text{exp}}$.
6. The final premise valides the expansion by checking the marked expansion against the type specified by the TSM, and generates the final expansion, e, according to the rules defining the *expansion validation judgements*:

Judgement Form	**Description**
$\Delta_{\text{out}}; \Delta \vdash \dot{\tau} \rightsquigarrow \tau\ \mathtt{type}$	Marked type $\dot{\tau}$ expands to τ under outer context Δ_{out} and current context Δ.
$\Delta_{\text{out}}\ \Gamma_{\text{out}}; \Delta\ \Gamma \vdash \dot{e} \rightsquigarrow e : \tau$	Marked expression $\dot{e}$ expands to e at type τ under outer contexts Δ_{out} and Γ_{out} and current contexts Δ and Γ.

   Each form in the syntax of expanded expressions has a corresponding form in the syntax of marked expressions (cf. Figure 3.2). For each rule in the static semantics of e, there is a corresponding expansion validation rule where the marked and expanded forms correspond. Only the current contexts are examined or extended by these rules. For example, the expansion validation rules for variables, functions and function application are shown below (the remaining such rules are analagous, but we omit them for concision):

$$\textsc{T-M-var}\quad \frac{}{\Delta_{\text{out}}\ \Gamma_{\text{out}};\Delta\ \Gamma, x:\tau \vdash x \rightsquigarrow x:\tau}$$

$$\textsc{T-M-abs}\quad \frac{\Delta_{\text{out}};\Delta \vdash \dot{\tau} \rightsquigarrow \tau\ \texttt{type} \qquad \Delta_{\text{out}}\ \Gamma_{\text{out}};\Delta\ \Gamma, x:\tau \vdash \dot{e} \rightsquigarrow e:\tau'}{\Delta_{\text{out}}\ \Gamma_{\text{out}};\Delta\ \Gamma \vdash \lambda x{:}\dot{\tau}.\dot{e} \rightsquigarrow \lambda x{:}\tau.e:\tau \rightharpoonup \tau'}$$

$$\textsc{T-M-app}\quad \frac{\Delta_{\text{out}}\ \Gamma_{\text{out}};\Delta\ \Gamma \vdash \dot{e}_1 \rightsquigarrow e_1:\tau \rightharpoonup \tau' \qquad \Delta_{\text{out}}\ \Gamma_{\text{out}};\Delta\ \Gamma \vdash \dot{e}_2 \rightsquigarrow e_2:\tau}{\Delta_{\text{out}}\ \Gamma_{\text{out}};\Delta\ \Gamma \vdash \dot{e}_1(\dot{e}_2) \rightsquigarrow e_1(e_2):\tau'}$$

The purpose of the outer contexts is to "remember" the context that the macro application appeared in so that spliced subexpressions extracted from the body, which are marked with the form $\mathsf{spliced}(\hat{e})$, can be checked appropriately:

$$\textsc{T-M-spliced}\quad \frac{\Delta_{\text{out}}\ \Gamma_{\text{out}} \vdash \hat{e} \rightsquigarrow e:\tau}{\Delta_{\text{out}}\ \Gamma_{\text{out}};\Delta\ \Gamma \vdash \mathsf{spliced}(\hat{e}) \rightsquigarrow e:\tau}$$

The current contexts are initially empty when checking the marked expansion generated by the parse function, so we achieve hygiene: the expansion cannot make any assumptions about the variables available in the outer context.

3.3.3 Metatheory

3.3.4 Static Language

3.4 Additional Conveniences

3.5 Limitations

Chapter 4

Pattern TSMs

TSMs as we have described them so far decrease the syntactic cost of introducing a value at a specified type. In full-scale functional languages like ML, one typically deconstructs a value using *nested pattern matching*. For example, let us return to the definition of the datatype `Rx` shown at the beginning of Sec. **??**. We can pattern match over a value, `r`, of type `Rx` using VerseML's **`match`** construct like this:

```
match r with
    Seq(Str(name), Seq(Str ": ", ssn)) => display name ssn
  |  _ => raise Invalid
```

In a functional language with primitive support for regular expression pattern syntax, we would expect to be able to write this example more concisely using the splicing forms discussed in Sec. 3.1:

```
match r with
    /@name: %ssn/ => display name ssn
  | _ => raise Invalid
```

Patterns are not expressions, so we cannot simply use a TSM defined at type `Rx` in a pattern. To address this, we must extend our language with support for typed pattern syntax macros (TPSMs). TPSMs are entirely analagous to TSMs, differing primarily in that the expansions they generate are patterns, rather than expressions. Assuming the abstract syntax of patterns is encoded by the type `Pat` (analagous to `Exp`), we can define a TPSM at type `Rx` as follows:

```
pattern syntax rx at Rx {
  static fn (body : Body) : Pat =>
    (* regex pattern parser here *)
}
```

Using this TPSM, we can rewrite our example as follows:

```
match r with
    rx /@name: %ssn/ => display name ssn
  | _ => raise Invalid
```

To ensure that the client of the TPSM need not "guess at" what variables are bound by the pattern, variables (e.g. `name` and `ssn` here) can only appear in spliced subpatterns (just as variables bound at the use site can only appear in spliced subexpressions when using TSMs). We leave a formal account of TPSMs (in a reduced calculus that features simple pattern matching) as work that remains to be completed (see Sec. **??**).

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. **??**).

Chapter 5

Parameterized TSMs

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 Formalism

A formal specification of TSLs in a language that supports only non-parametric datatypes is available in a paper published in ECOOP 2014 [28]. We will add support for parameterized TSLs in the dissertation (see Sec. **??**).

Chapter 7

Conclusion & Future Directions

7.1 Summary

TODO: Write summary

7.2 Future Directions

7.2.1 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 [24]) 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

7.2.2 Improved Error Reporting

7.2.3 Type-Aware Splicing

7.2.4 Integration With Code Editors

7.2.5 Resugaring

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

7.2.6 Non-Textual Display Forms

TODO: Talk about active code completion work and future ideas

Bibliography

TODO (Later): List conference abbreviations.

TODO (Later): Remove extraneous nonsense from entries.

[1] The Visible Compiler. `http://www.smlnj.org/doc/Compiler/pages/compiler.html`. 2.2.3, 3.1.2, 5

[2] Information technology portable operating system interface (posix) base specifications, issue 7. *Information technology Portable Operating System Interface (POSIX) Base Specifications, Issue 7*. 2.1.1

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

[4] 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`. 2.2.3

[5] 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`. 2.2.1

[6] 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. 2.2.3

[7] 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

[8] 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

[9] 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`. 2.1.1

[10] Sebastian Erdweg and Felix Rieger. A framework for extensible languages. In *GPCE '13*,

pages 3–12, 2013. 1.1, 2.2.2

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

[12] 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, 2.2.2

[13] 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. 2.2.3

[14] 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. 2.2.2

[15] 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

[16] Robert Harper. Programming in Standard ML. `http://www.cs.cmu.edu/~rwh/smlbook/book.pdf`. Retrieved June 21, 2015., 1997. 1.1

[17] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012. 2.1.1, 3.1.2, 3.3.1

[18] T. P. Hart. MACRO definitions for LISP. Report A. I. MEMO 57, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, October 1963. 2.2.3

[19] David Herman. *A Theory of Typed Hygienic Macros*. PhD thesis, Northeastern University, Boston, MA, May 2010. 2.2.3

[20] 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

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

[22] 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. 2.2.3

[23] 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, 2.2.2

[24] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`. Version 8.0. 2.2.2, 7.2.1

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

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

[27] 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

[28] 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

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

[30] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. 3.3.1

[31] 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

[32] 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

[33] 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, 2.2.2

[34] Denys Shabalin, Eugene Burmako, and Martin Odersky. Quasiquotes for Scala. Technical Report EPFL-REPORT-185242, 2013. 2.2.3

[35] 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

[36] Guy L Steele. *Common LISP: the language*. Digital press, 1990. 2.2.2

[37] 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`. 2.1.1

[38] Todd L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University, 2004. 1.2

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

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