Modularly Programmable Syntax (Thesis Proposal)

Cyrus Omar
Computer Science Department
Carnegie Mellon University
comar@cs.cmu.edu

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 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.

1 Motivation

Functional programming language designers have long studied small typed lambda calculi to develop a principled understanding of language-theoretic issues, like type safety, and to examine the mathematical character of language primitives of interest in isolation. These studies have then informed the design of "full-scale"[1] functional languages, which combine several such primitives and also introduce various generalizations and primitive "embellishments" motivated by a consideration of human factors. For example, major functional languages like Standard ML (SML) [25, 16], OCaml [23] and Haskell [21] all primitively build in record types, generalizing the nullary and binary product types that suffice in simpler calculi, because explicitly labeled components are cognitively useful to human programmers. Similarly, these languages build in derived syntactic forms (colloquially, "syntactic sugar") that decrease the syntactic cost of working with common library constructs, like lists.

The hope amongst many language designers is that a limited number of primitives like these will suffice to produce a "general-purpose" programming language, i.e. one where programmers can direct their efforts almost exclusively toward the development of libraries for a wide variety of application domains. However, a stable language design that fully achieves this ideal has yet to emerge, as evidenced by the diverse array of "dialect" languages that continue to proliferate around all major contemporary languages. In fact, tools that aid in the construction of so-called "domain-specific" language dialects (DSLs)[2] seem only to be becoming increasingly prominent.

Although some of these dialects extend the type structure of the language that they are based on, the most common situation, and the one that will be our focus in this work, is

[1] Throughout this work, words and phrases that should be read as having an intuitive or informal meaning, rather than a strict mathematical meaning, will be introduced with quotation marks.

[2] In some parts of the literature, such dialects are called "external DSLs", to distinguish them from "internal" or "embedded DSLs", which are actually library interfaces that only "resemble" distinct dialects [12].

that the existing semantic primitives suffice and it is only the *syntactic cost* of expressing one or more constructs of interest using these primitives that is too high. In response, library providers construct *syntactic dialects* – dialects that introduce only new derived syntactic forms. For example, Ur/Web is a syntactic dialect of Ur (a language that itself descends from ML [7]) that builds in derived forms for SQL queries, HTML elements and other datatypes used in the domain of web programming [8]. This is not an isolated example – there are many other types of data that could similarly benefit from the availability of specialized derived forms (we will consider another example, regular expression patterns, in Sec. 3.1). Tools like Camlp4 [23], Sugar* [9, 10] and Racket [11], which we will discuss in Sec. 3.2, have lowered the engineering costs of constructing syntactic dialects in such situations, further contributing to their proliferation.

1.1 Dialects Considered Harmful

Some view the ongoing proliferation of dialects described above as harmless or even as desirable, arguing that programmers can simply choose the right dialect for the job at hand [37]. However, this "dialect-oriented" approach is, in an important sense, anti-modular: a library written in one dialect cannot, in general, safely and idiomatically interface with a library written using another dialect. Addressing this interoperability problem requires somehow "combining" the dialects into a single language [4]. However, in the most general setting where the dialects in question might be specified by judgements of arbitrary form, this is not a well-defined notion. Even if we restrict our interest to dialects specified using formalisms that do operationalize some notion of dialect combination, there is generally no guarantee that the combined dialect will conserve important metatheoretic properties that can be established about the dialects in isolation. For example, consider two syntactic dialects, one specifying derived syntax for finite mappings, the other specifying a similar syntax for *ordered* finite mappings. Though each dialect can be specified by an unambiguous grammar in isolation, when these grammars are naïvely combined by, for example, Camlp4, ambiguities arise. Due to this paucity of modular reasoning principles, the "dialect-oriented" approach is problematic for software development "in the large".

Dialect designers have instead had to take a less direct approach to have an impact on large-scale software development: they have had to convince the designers in control of comparatively popular languages, like OCaml and Scala, to include some suitable variant of the primitives they've espoused into backwards compatible language revisions. This *ad hoc* approach is not sustainable, for three main reasons. First, there are simply too many potentially useful such primitives, and many of these are only relevant in relatively narrow application domains (for derived syntax, our group has gathered initial data speaking to this [26]). 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 replaced without breaking backwards compatibility.

This suggests that language designers should strive 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 further "embellishments" and settle on the existing primitives, which might be considered to sit at a "sweet spot" in the overall language design space. 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 dialects today instead to modularly composable 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 [38]. 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. However, syntactic machinery remains built in.

2 Proposed Contributions

Our aim in the work being proposed is to introduce primitive language constructs that take further steps down the second path just described by reducing the need for syntactic dialects. In particular, we plan to introduce the following primitives:

1. **Typed syntax macros** (TSMs), introduced in Sec. 4, reduce the need to primitively build in derived concrete syntactic forms specific to library constructs (e.g. list syntax as in SML or XML syntax as in Scala and Ur/Web), by giving library providers static control over the parsing and expansion of delimited segments of textual syntax (at a specified type or parameterized family of types).

2. **Type-specific languages** (TSLs), introduced in Sec. 5, 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 implicitly.

As vehicles for this work, we plan to formally specify a series of small typed lambda calculi that capture each of the novel primitives that we introduce "minimally". For the sake of examples, we will also describe (but not formally specify) a "full-scale" functional language called Verse.[3] Verse is based semantically on Standard ML, differing mainly in that it uses a local type inference scheme [30] (like, for example, Scala) for reasons that we will return to in Sec. 5. The reason we will not follow Standard ML [25] in giving a complete formal specification of Verse is both to emphasize that the primitives we introduce can be considered for inclusion in a variety of language designs, and to avoid distracting the reader with specifications for "orthogonal" primitives that are already well-understood in the literature.

The primitives we propose 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 static representations of terms. Static functions are functions written in a restricted subset of the language (we will discuss the design space of restrictions in Sec. 4.3). 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 aspects of the language (but did not take a type theoretic approach) [36].

The main challenge in the design of these primitives 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.[4] Our main technical contributions will be in rigorously showing how to address these problems in a principled manner. In particular, syntactic conflicts will be impossible by construction and the semantics will validate code statically generated by TSMs and TSLs to maintain:

- a *hygienic type discipline*, meaning that the language is type safe, that one can reason about the type of a well-typed expression without examining its expansion, and that the expansion does not make any assumptions about the names of variables in the surrounding context; 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.[5]

[3] We distinguish Verse 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.

[4] This is why languages like Verse 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.

[5] 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.

2.1 Thesis Statement

In summary, we propose a thesis defending the following statement:

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

2.2 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 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.

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, Verse will come with a standard library defining a carefully 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 the potential for "abuse"). For most programmers, using Verse should not be substantially different from using a language like ML or one of its dialects.

Finally, Verse intentionally is not a dependently-typed language like Coq, Agda or Idris, because these languages do not maintain a phase separation between "compile-time" and "run-time." This phase separation is useful for programming tasks (where one would like to be able to discover errors before running a program, particularly programs that may have an effect) but less so for theorem proving tasks (where it is mainly the fact that a pure expression is well-typed that is of interest, by the propositions-as-types principle). Verse is designed to be used for programming tasks where SML, OCaml, Haskell or Scala would be used today, not for advanced theorem proving tasks. That said, we conjecture that the primitives we describe could be added to languages like Gallina (the "external language" of the Coq proof assistant [24]) with modifications, but do not plan to pursue this line of research in this dissertation.

3 Background

Verse, like most contemporary languages, specifies a textual concrete syntax.[6] Because the purpose of this syntax is to serve as the programmer-facing user interface of the language, it is common practice to build in derived syntactic forms (colloquially, *syntactic sugar*) that capture common idioms more concisely or "naturally". For example, derived list syntax is built in to many functional languages, so that instead of having to write out `Cons(1, Cons(2, Cons(3, Nil)))`, the programmer can equivalently write `[1, 2, 3]`. Many languages go beyond this, building in derived syntax associated with various other types of data, like vectors (the SML/NJ dialect of SML), arrays (OCaml), monadic commands (Haskell), syntax trees (Scala, F#), XML documents (Scala, Ur/Web) and SQL queries (F#, Ur/Web).

Verse takes a less *ad hoc* approach – rather than privileging particular library constructs with primitive syntactic support, Verse exposes primitives that allow library providers to introduce new expansion logic on their own, in a safe and modular manner.

We will begin in Sec. 3.1 by detailing another example for which such a mechanism would be useful: regular expression (regex) patterns expressed using abstract data types. We will refer to this example throughout the proposal. In Sec. 3.2, we discuss how the

[6] Although Wyvern specified a layout-sensitive concrete syntax, to avoid unnecessary distractions, we will describe a more conventional layout-insensitive concrete syntax for Verse.

usual approach of using dynamic string parsing to introduce regex patterns is not ideal. We also survey existing alternatives to dynamic string parsing, finding that they involve an unacceptable loss of modularity and other undesirable trade-offs. In Secs. 4 and 5, we introduce our proposed alternatives – *typed syntax macros* (TSMs) and the related *type-specific languages* (TSLs) – and discuss how they resolve these issues (as well as some limitations that they have). We also give an overview of how TSMs are formally specified. We give a concrete timeline for the remaining work in Sec. 6, and conclude in Sec. 7.

3.1 Motivating Example: Regular Expression Syntax

Let us begin by taking the perspective of a regular expression library provider. We assume the reader has some familiarity with regular expressions [35] and Standard ML [16]. We will discuss a standard variant of regular expressions that supports marking *captured groups* (with parentheses in the concrete syntax) to make certain examples more interesting.

Abstract Syntax The abstract syntax of patterns, r, over strings, s, is specified below:

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

One way to express this abstract syntax is by defining a recursive sum type [17]. Verse supports these as datatypes:

```
datatype Rx {
    Empty | Str of string | Seq of Rx * Rx
    | Or of Rx * Rx | Star of Rx | Group of Rx
}
```

However, there are some reasons not to expose this representation of patterns directly to clients. First, regular expression patterns are usually identified up to their reduction to a normal form. For example, $\mathbf{seq}(\mathbf{empty}, r)$ has normal form r. It might be useful for patterns 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 automata) without exposing this "implementation detail" to clients. Indeed, there may be many ways to represent regular expression patterns, each with different performance trade-offs. For these reasons, a better approach in Verse, as in ML, is to abstract over the choice of representation using the module system's support for type abstraction. In particular, we can define the following *module signature*, 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
}
```

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).

Concrete Syntax The abstract syntax of patterns is too verbose to be practical in all but the most trivial examples, so programmers conventionally write patterns using a more concise concrete syntax. For example, the concrete syntax `A|T|G|C` corresponds to the following much more verbose pattern expression (assuming some module `R : RX` is in scope here and in the remainder of the document):

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

3.2 Existing Approaches

3.2.1 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 escape character`[7], 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 [28]. 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:

[7] This is the error message that `javac` produces. When compiling an analagous expression using SML of New Jersey (SML/NJ), we encounter a more confusing error message: `Error: unclosed string`.

```
fun 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 [2]. 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" [33].

   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).[8] 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. 3.2.2 and 3.2.3 respectively.

3.2.2 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* [34] (which are distinct from its term-rewriting macros, described in Sec. 3.2.3 below) and Racket's preprocessor [11]. 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

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

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 [31]. 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_verse_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_verse_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.

3.2.3 Static 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 [19, 13] 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 [3] and Scala [32], 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.

4 Typed Syntax Macros (TSMs)

We will now introduce a new primitive – the **typed syntax macro** (TSM) – that combines the syntactic flexibility of syntax extensions with the reasoning guarantees of typed macros. Like the term-rewriting macros just described, TSMs can be parameterized by modules, so they can be used to define syntax valid at any abstract type defined by a module satisfying a specified signature. As we will discuss in the remainder of this section, this addresses all of the problems brought up above, at moderate syntactic cost.

4.1 TSMs By Example

To introduce TSMs, consider the following concrete external expression:

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

Here, we apply a *parameterized TSM*, `rx`, first to a module parameter, `R`, then to a *delimited form*, `/A|T|G|C/`. A number of alternative delimiters are also available in Verse's concrete syntax and could equivalently have been used, e.g. `<A|T|G|C>`. The TSM statically

parses the *body* of the provided delimited form, i.e. the characters between the delimiters (indicated here in blue), and computes an *expansion*, i.e. another expression. In this case, **pattern** generates the following expansion (written concretely):

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

The definition of `rx` looks like this:

```
syntax rx[Q : RX] at Q.t {
  static fn (body : Body) : Exp => (* regex expression parser here *)
}
```

This *TSM definition* first identifies the TSM as `rx`, then specifies a module parameter, `Q`, which quantifies over modules that match the signature `RX` (again, we use `Q` to emphasize that the library provider does not make any assumptions about identifiers in scope; the client must pass in a particular module, e.g. `R` above). The *type annotation* **at** `Q.t` specifies that all expansions that arise from the application of this TSM to a module, `Q`, and a delimited form must necessarily be of type `Q.t`.

When a client applies the TSM to the necessary parameters and a delimited form, e.g. the parameter `P` and the delimited form `/A|T|G|C/` in the example above, the expansion is computed by calling the *parse function* defined within braces. The parse function is a static function of type `Body -> Exp`. Both of these types are defined in the Verse *prelude*, which is a set of definitions available ambiently. The type `Body` gives the static function access to the body of the delimited form (e.g. the characters in blue above) and the type `Exp`, mentioned also in the discussion of macros above, encodes the abstract syntax of expressions (there are also encodings of other syntactic classes, e.g. types and variables, that can appear in expressions).

To support splicing syntax as described in Sec. 3.2.2, the parse function must be able to extract subexpressions directly from the supplied body. For example, consider the client code below:

```
(* TSMs can be partially applied and abbreviated *)
let syntax rx = rx R
let val ssn = rx /\d\d\d-\d\d-\d\d\d\d/
fun example_using_tsm(name: string) =
  rx /@name: %ssn/
```

The subexpressions `name` and `ssn` on the last line appear directly in the body of the delimited 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 `@` and `%` followed by an identifier), it can mark this subsequence as such. Such marked subsequence can appear as subexpressions directly within the expansion being generated (i.e. marked subsequences of the body are of type `Exp`). For example, the expansion generated for the body of `example_using_tsm` would, if written concretely with spliced expressions marked, be:

```
Q.Seq(Q.Str(spliced<name>), Q.Seq(Q.Str ": ", spliced<ssn>))
```

Hygiene is achieved by checking that only these marked portions of the generated expansion refer to the variables at the use site, preventing inadvertent variable capture by the expansion. In other words, the TSM cannot make any assumptions about variable names at the use site.

After checking that the expansion is valid, i.e. that it is hygienic as just described and that the expansion is of the type specified by the TSM, the semantics removes the splicing markers just mentioned and removes the abstraction barrier by substituting the actual module parameter `R` for `Q`, producing the final expansion, as expected:

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

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
$\eta ::= m \mid \mathsf{syntax} \mathbin{@} \tau \, \{\hat{e}\}$

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

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

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

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

Figure 1: Syntax of $\mathcal{L}^{\text{TSM}}$

4.2 Formalization

To give a formal account of TSMs, we will now introduce a typed lambda calculus, $\mathcal{L}^{\text{TSM}}$. To simplify matters, this calculus supports only unparameterized TSMs (i.e. TSMs defined at a single type, e.g. the datatype `Pattern`, rather than a parameterized family of types). In the dissertation, we will introduce a second calculus, $\mathcal{L}^{\text{PTSM}}$, that extends $\mathcal{L}^{\text{TSM}}$ with support for parameterized families of types as described above (see Sec. 6). The syntax of $\mathcal{L}^{\text{TSM}}$ is shown in Figure 1.

4.2.1 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* [29] or *PFPL* [17]). We will reproduce a more detailed account of the 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`.

4.2.2 Macro Expansion

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 1). 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):

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

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

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

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:

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

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

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

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:

(T-U-TSM-ap)
$$\frac{\begin{array}{rl}\Delta \vdash \tau\ \texttt{type} & \emptyset\ \emptyset \vdash \hat{e}_{\text{parse}} \rightsquigarrow e_{\text{parse}} : \mathsf{Body} \rightharpoonup \mathsf{Exp} \\ b \downarrow e_{\text{body}} : \mathsf{Body} & e_{\text{parse}}(e_{\text{body}}) \Downarrow e_{\text{exp}} \qquad e_{\text{exp}} : \mathsf{Exp} \uparrow \dot{e}_{\text{exp}} \\ \multicolumn{2}{c}{\Delta\ \Gamma; \emptyset\ \emptyset \vdash \dot{e}_{\text{exp}} \rightsquigarrow e : \tau}\end{array}}{\Delta\ \Gamma \vdash \mathsf{syntax}\ @\ \tau\ \{\hat{e}_{\text{parse}}\}\ /b/ \rightsquigarrow e : \tau}$$

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 1, 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 : Var $\uparrow x$	e decodes to variable x.
e : TVar $\uparrow t$	e decodes to type variable t.
e : Type $\uparrow \dot{\tau}$	e decodes to marked type $\dot{\tau}$.
e : 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**
$\Delta_{\text{out}}; \Delta \vdash \dot{\tau} \rightsquigarrow \tau$ `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 1). 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):

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

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

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

   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 spliced$(\hat{e})$, can be checked appropriately:

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

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.

4.3 Limitations

Though expansion validation ensures that an incorrect parser cannot threaten type safety, proving type safety inductively is subtle because the typed expansion judgement and the expansion validation judgement are mutually defined. We must come up with a decreasing metric to ensure that induction is well-founded. We will formally define such a metric in the dissertation, but informally, notice that we only induct on spliced terms derived from the body, which is itself a subterm of the original unexpanded expression.

To provide the strongest metatheoretic guarantees, we must have that the static subset of the language is total. Allowing non-termination in the parse function (as we have done in the formalism above, by not distinguishing a static subset at all) would cause typed expansion to become undecidable (typechecking expanded expressions would remain decidable, however). Related concerns would arise if the static subset supported external (e.g. I/O) effects – one would generally not want the mere act of running the typechecker on a program to have an arbitrary effect on, for example, the programmer's file system. Mutation effects could also be problematic if we allow static values to be "promoted" into expansions (a feature we did not specify above, but will return to in the dissertation).

Another limitation is that TSMs as we have described them only capture idioms that occur within a single parameterized family of types, not idioms that might span many (or all) types, e.g. control flow idioms. In fact, this is intentional – neither humans nor tools need to examine the expansion to determine what type it must necessarily have. We will, however, discuss other points in the design space in the dissertation (based on some of the work in a recent paper we have published [27]).

4.4 Typed Pattern Syntax Macros (TPSMs)

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. 3.1. We can pattern match over a value, `r`, of type `Rx` using Verse'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. 4.1:

```
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. 6).

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. 3.1. We leave further discussion of such a facility and of parameterized TPSMs also as remaining work (see Sec. 6).

5 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 Verse 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.

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.

5.1 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 *)
```

5.2 Formalism

A formal specification of TSLs in a language that supports only non-parametric datatypes is available in a paper published in ECOOP 2014 [26]. We will add support for parameterized TSLs in the dissertation (see Sec. 6).

6 Timeline and Milestones

We described and gave a more detailed formal specification of TSMs in a recently published paper [27], and TSLs in a paper published last year [26], both in the context of the Wyvern programming language. The mechanics of defining a TSM, statically invoking the parse function, expansion encoding/decoding and hygienic expansion validation have all been detailed there.

In the dissertation, I plan to first present the formal system $\mathcal{L}^{\text{TSM}}$. The account in Sec. 4.2 represents the first and most substantial step of this plan. This is complete on paper, so I only need to typeset the omitted details, including the metatheory. I plan to finish this by the **end of October**. This system will serve as the basis for several extensions that capture

other constructs described above, so I will ask the committee to read this section in detail once it is complete.

The first extension will add support for simple TSLs over unparameterized datatypes. This has already been written up in [26], so I plan to have this in final form in the **first week of November**.

The next extension will add support for pattern matching over these datatypes, and TPSMs. I will base this extension on the account of pattern matching given in *PFPL* [17]. I plan to have a rough sketch of how this should work by **late-November**, in consultation with the committee, and a final write-up of it by **mid-December**.

The final extension will add support for type and module parameters. To do so, I will assume that the available module paths and their signatures are being tracked by a module context in the static semantics, without detailing how the module language populates this context (i.e. I will only specify the "core language"). I plan to sketch this out and discuss it with my committee also **during November** and finish writing the details up by the **last week of December**.

Over the course of the **Spring semester**, I plan to write the remaining sections (e.g. additional background, related and future work sections, and additional examples), flesh out the proofs of minor lemmas and respond to the committee's requests for edits. Based on this timeline, I plan to defend before the **end of the Spring semester**.

My current draft of the dissertation document will be available on GitHub:
`https://github.com/cyrus-/thesis/blob/master/omar-thesis.pdf`

7 Conclusion

In summary, we have proposed a new sort of macro that gives programmers the ability to programmatically express new syntactic expansions using statically evaluated functions. These expansions are hygienic and restricted to a single parameterized family of types, so that programmers can more easily read and reason about programs that use them. By using local type inference, the syntactic cost of using these syntactic expansions is kept low. Syntactic conflicts are impossible by construction. These macros also interact well with features found in typed functional languages like ML, including datatypes, modules and pattern matching. Consequently, Verse does not need to build in syntactic constructs that comparable languages need to (or would need to) build in, like regular expression syntax, list syntax and HTML syntax.

We have also outlined a formal semantics for these novel primitives. A more detailed specification and metatheoretic results that validate our claims that Verse has a hygienic type discipline and strong modular reasoning principles represent the main avenues for remaining work. We have already made substantial progress on closely related variants of the mechanisms proposed here, so we anticipate that the work can be completed over the remainder of the semester, per the timeline above.

References

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

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

[3] A. Bawden. Quasiquotation in Lisp. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 4–12, 1999.

[4] N. Benton and A. Kennedy. Interlanguage Working Without Tears: Blending SML with Java. In *ICFP '99*, pages 126–137, 1999.

[5] M. Bravenboer, E. Dolstra, and E. Visser. Preventing Injection Attacks with Syntax Embeddings. In *GPCE '07*, pages 3–12, 2007.

[6] E. 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.

[7] A. Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In B. G. Zorn and A. 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.

[8] A. Chlipala. Ur/Web: A simple model for programming the web. In *POPL '15*, pages 153–165, 2015.

[9] S. Erdweg, T. Rendel, C. Kastner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *OOPSLA '11*, pages 187–188, 2011.

[10] S. Erdweg and F. Rieger. A framework for extensible languages. In *GPCE '13*, pages 3–12, 2013.

[11] M. Flatt. Creating languages in Racket. *Commun. ACM*, 55(1):48–56, Jan. 2012.

[12] M. Fowler and R. Parsons. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[13] S. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *ICFP '01*, pages 74–85, 2001.

[14] T. Griffin. Notational definition-a formal account. In *Logic in Computer Science (LICS '88)*, pages 372–383, 1988.

[15] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, Mar. 1996.

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

[17] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.

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

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

[20] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

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

[22] E. E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Symposium on LISP and Functional Programming*, pages 151–161, Aug. 1986.

[23] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system release 4.01 Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, September 2013.

[24] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[25] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[26] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely composable type-specific languages. In *ECOOP '14*, 2014.

[27] C. Omar, C. Wang, and J. Aldrich. Composable and hygienic typed syntax macros. In *ACM Symposium on Applied Computing (SAC '15)*, 2015.

[28] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active Code Completion. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE '12)*, pages 859–869, 2012.

[29] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[30] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, Jan. 2000.

[31] A. Schwerdfeger and E. V. Wyk. Verifiable composition of deterministic grammars. In *PLDI '09*, pages 199–210, 2009.

[32] D. Shabalin, E. Burmako, and M. Odersky. Quasiquotes for Scala. Technical Report EPFL-REPORT-185242, 2013.

[33] E. Spishak, W. Dietl, and M. 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.

[34] G. L. Steele. *Common LISP: the language*. Digital press, 1990.

[35] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[36] T. L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University, 2004.

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

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