Modularly Metaprogrammable Syntax and Type Structure (Thesis Proposal)

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

Abstract

Functional programming languages like ML descend conceptually from simple lambda calculi, but to be pragmatic, must expose a syntax and type structure to programmers of a more elaborate design. Language designers have many viable choices at this level, as evidenced by the diversity of dialects that have proliferated around all major languages. But language dialects cannot, in general, be modularly composed, limiting the choices available to programmers. We propose a new language, Verse, designed to decrease the need for dialects by giving library providers the ability to safely and modularly express new derived syntax and external type structure of a variety of designs, atop a minimal typed lambda calculus, called the Verse *internal language*.

1 Motivation

Well-designed general-purpose programming languages give library providers the power to modularly express a wide variety of useful constructs using a comparatively small set of primitives. There has been substantial progress on many of the foundational semantic issues that are relevant to the design of general-purpose languages over the past decades, much of it driven by advances in type theory. However, when more pragmatic language design issues are brought into consideration, a broadly agreeable set of primitives remains elusive, as evidenced by the diverse array of dialects that surround around all major contemporary languages. For example, let us consider the many dialects of ML. Though they generally agree on the importance of a powerful module system atop a core language based conceptually on the polymorphic lambda calculus with sums, products and recursive types, they all introduce various syntactic and semantic extensions and tweaks of their own novel design (we discuss a number of examples below). Tools for constructing new "domain-specific" language dialects also continue to proliferate. Why are well-informed programmers and researchers so often tempted to construct dialects of a language like ML?

Perhaps the most common reason is simply that, for human programmers, the *syntactic cost* of expressing a construct of interest using general-purpose primitives is not always ideal. For example, we will consider regular expression patterns expressed using abstract data types (in ML, a mode of use of the module system) in Section 4. This gives us precisely the semantics that we need, but the syntactic cost is not quite ideal. In situations like this, library providers are often tempted to construct a *derived syntactic dialect*, i.e. a dialect specified by context-free elaboration to the existing language, that introduces derived syntax specific to their library. For example, Ur/Web is a derived syntactic dialect of Ur (itself descended from ML) that builds in derived syntax for SQL queries, HTML elements and other datatypes common in web programming [6]. Indeed, nearly all major languages primitively build in derived syntax for constructs that are otherwise expressed in the standard library, e.g. derived list syntax in Standard ML (SML) [13, 22]. Tools like

Camlp4 [18] and Sugar* [7, 8], which we will return to later, help lower the engineering costs of constructing derived syntactic dialects, contributing to their proliferation.

More advanced dialects introduce new type structure, going beyond what is possible by a context-free elaboration to an existing language. For example, SML goes beyond the simple nullary and binary product types common in simpler calculi, instead exposing record types. Other dialects have explored "record-like" primitives that support functional update and extension operators, width and depth coercions (sometimes implicit), mutable rows, methods, prototypic dispatch and various other embellishments. Ocaml builds in a more sophisticated object system, polymorphic variants and conveniences like logic for typechecking operations that use format strings, e.g. `sprintf` [18]. ReactiveML builds in primitives for functional reactive programming [20]. ML5 builds in primitives for distributed programming [23]. Manticore builds in parallel arrays [9]. Alice ML builds in futures and promises [29]. MLj builds in essentially the entire Java programming language, to support typesafe interoperation with existing Java code [3]. Indeed, perusal of the proceedings of any major programming languages conference will typically yield several more examples. Tools like proof assistants and logical frameworks are designed to support specifying and reasoning about dialects like these, and tools like compiler generators and language workbenches lower the engineering cost of implementing them.

The problem with this practice of using language dialects as vehicles for new syntactic and semantic primitives is that it is, in an important sense, anti-modular: a program written in one dialect cannot, in general, safely and idiomatically interface with libraries written in a different dialect. This would fundamentally require combining the two dialects into a single language. Left unconstrained, this is an ill-posed problem, i.e. given two dialects that are specified by judgements of arbitrary form, it is not clear what it would even mean to combine them. Even when the dialects are specified using a formalism that does provide some way to naïvely combine two languages, there is generally no guarantee that the composition will conserve important syntactic and semantic properties. For example, consider two derived syntactic dialects, one building in derived syntax for JSON (a popular data interchange format), the other using a similar syntax for finite mappings of some other type. Though each is known to have an unambiguous concrete syntax in isolation, when their syntax is naïvely combined (e.g. by Camlp4), ambiguities arise. Any putative "combined language" must formally be considered a distinct system for which one must derive all metatheorems of interest anew, guided only informally by those derived for the dialects individually. Due to this paucity of modular reasoning principles, language dialects are not practical artifacts for software development "in the large".

Dialects do, however, have an indirect impact on the practice of programming because the language designers that control comparatively popular languages, like Ocaml and Scala, can occasionally be convinced to incorporate ideas from dialects into backwards compatible language revisions. These major languages have thus become more expressive over time. However, this has come at a cost: they have also ballooned in size. Moreover, because there appears to be a "long tail" of potentially useful primitives (for derived syntactic primitives, our group has gathered initial data speaking to this [24]), primitives that are only situationally useful, or that trigger aesthetic disagreements between different factions of the community, are still often left languishing in "toy" dialects. A conservative approach to incorporating new primitives is only sensible because once a primitive is introduced, it becomes entrenched as-is and monopolizes "syntactic resources", as we will discuss below. Recalling the words of Reynolds, which are clearly as relevant today as they were nearly half a century ago [28]:

> *The recent development of programming languages suggests that the simultaneous achievement of simplicity and generality in language design is a serious unsolved problem.*

There are two possible paths forward for the subset of the language design community interested in keeping general-purpose languages small and free of *ad hoc* primitives. One, exemplified (arguably) by Standard ML, is to generally eschew the introduction of new primitives and settle on a set of primitives that sit at a "sweet spot" in the overall language design space, accepting that in some circumstances, these primitives trade away expressive power or have high syntactic cost. The other path forward is to press on in the search for even more general language primitives, i.e. primitives that are so expressive that they allow us to degrade a broad class of existing primitives, including those found in dialects, into modularly composable library constructs, where they can be evaluated on their individual merits by programmers. Such primitives do sometimes arise. For example, Ocaml recently introduced "generalized algebraic data types" (GADTs), based on research on guarded recursive datatype constructors [34]. Using GADTs, Ocaml was able to move some machinery for typechecking operations that use format strings, like `printf`, out of the language and into the standard library (though some machinery remains built in). Our broad aim will be to introduce a new general-purpose programming language called Verse[1] that takes even more dramatic steps down this second path.

2 Proposed Contributions

Verse has a module system taken directly from SML, but its core language is organized in a novel manner around a minimal typed lambda calculus, called the Verse *internal language*, introduced in Section 3. In lieu of typical external (i.e. programmer-facing) derived syntax and type structure, Verse introduces two novel primitive constructs: **typed syntax macros** (TSMs), in Section 4, and **metamodules**, in Section 5. Briefly summarized, TSMs subsume the need for derived syntax specific to library constructs (e.g. list syntax in SML) by giving library providers static control over the parsing and elaboration of delimited segments of textual concrete syntax. Metamodules give library providers static hooks directly into the semantics of the Verse EL, allowing library providers to, for example, define the type structure of records (and various "record-like" constructs, as discussed above) by type-directed translation to the internal language (which builds in only nullary and binary products). Both TSMs and metamodules can be understood as novel *metaprogramming* primitives, because they involve static code generation. We will also introduce a simple variant of each of these primitives that leverages Verse's support for local type inference to further reduce syntactic cost in certain common situations.

The key challenge in the design of these primitives will come in ensuring that they are metatheoretically well-behaved, given that they aim to give library providers decentralized control over aspects of the language that have typically been under the monolithic control of the language designer. If we are not careful, many of the problems mentioned above as inherent to combining distinct language dialects could simply manifest themselves in the semantics of these primitives.[2] Our main technical contributions will come in showing how to address these problems in a principled manner. In particular, syntactic ambiguities will be impossible by construction and we will validate the code statically generated by TSMs and metamodules to maintain a hygienic type discipline and, most uniquely, powerful modular reasoning principles. Library providers will be able to reason about the constructs that they have defined in isolation, and library clients will be able to use them safely in any combination, without the possibility of conflict.[3]

[1]We distinguish Verse from Wyvern, which is the language referred to in prior publications about some of the work being proposed here, because Wyvern is a group effort evolving independently in some important ways.

[2]This is why languages like Verse are often called *extensible languages*, though this is somewhat of a misnomer. The chief 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]We will assume that simple naming conflicts can be avoided extrinsically in some suitable manner, 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 programming language can give library providers the power to meta-programmatically express new derived syntax and external type structure atop a minimal typed internal language, using primitives that maintain a hygienic type discipline and modular reasoning principles. These primitives are expressive enough to subsume the need for a variety of primitives that are, or would need to be, built in to comparable contemporary languages.

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 dialects.

It may also be useful to explicitly acknowledge that library providers could leverage the primitives we introduce to define constructs that are in rather 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 primitives we introduce (following the example of languages that support operator overloading or typeclasses, which also have the potential for "abuse").

Finally, we are not interested here in languages that feature full-spectrum dependent types, which blur the phase separation between compile-time and run-time, though we conjecture that the primitives we introduce could be introduced into languages like Gallina (the "external language" of the Coq proof assistant) with some modifications. Verse should be compared to languages that maintain a phase separation, like ML and Scala.

3 Verse

Let us begin by briefly summarizing how Verse is organized and specified. Verse consists of a *module language* and a *core language*. The module language is based directly on SML's module language, which we assume familiarity with for the purposes of this work [13, 19]. We will give an example of its use in Section 4, but do not plan to specify it in detail.

The core language is organized much like the first stage of a type-directed compiler (e.g. the TIL compiler for Standard ML [32]), consisting of a user-facing *typed external language* (EL) specified by type-directed translation to a minimal *typed internal language* (IL). The main judgements in the specification of the EL take the following form (we omit mention of various contexts for now):

Judgement	**Pronunciation**
$e \Rightarrow \sigma \rightsquigarrow \iota$	External expression e synthesizes external type σ and has translation ι.
$e \Leftarrow \sigma \rightsquigarrow \iota$	External expression e analyzes against external type σ and has translation ι.
$\sigma\ \texttt{type} \rightsquigarrow \tau$	Static expression σ is an external type with translation τ.

Note that the typing judgements are *bidirectional*, i.e. there is a distinction between *type synthesis* (the type is an "output") and *type analysis* (the type is an "input") [27]. This is to allow us to explicitly specify how Verse's *local type inference* works, and in particular, how it interacts with the primitives that we will introduce. Like Scala, we intentionally do not aim to support global type inference, both because it is not compatible with the primitives we introduce as described and because we believe that in practice, the cognitive cost of the error messages that arise outweigh its marginal utility relative to local type inference. Note also that, for concision, we will write "expression" and "type" without qualification to refer to external expressions and types, respectively.

Our design is largely insensitive to the details of the IL. The only strict requirements are that the IL is type safe and supports parametric type abstraction. For our purposes, we will keep things simple by using the strictly evaluated polymorphic lambda calculus with nullary and binary product and sum types and recursive types, which we assume the reader has familiarity with and for which all the necessary metatheorems are well-established (we follow [14]). The main judgements in the static semantics of the IL (again omitting contexts for now) thus take the following form:

Judgement	**Pronunciation**
$\iota : \tau$	Internal expression ι has internal type τ.
τ `type`	Internal type τ is valid.

The dynamic semantics can be specified as a transition system with judgements of the following form:

Judgement	**Pronunciation**
$\iota \mapsto \iota'$	Internal expression ι transitions to ι'.
ι `val`	Internal expression ι is a value.

The iterated transition judgement $\iota \mapsto^* \iota'$ is the reflexive, transitive closure of the transition judgement, and the evaluation judgement $\iota \Downarrow \iota'$ is derivable iff $\iota \mapsto^* \iota'$ and ι' `val`.

Primitive features like state, exceptions, concurrency primitives, scalars, arrays and others characteristic of a first-stage compiler IL would also be included in practice, and in some cases this would affect the shape of the internal semantics, but we omit them when their inclusion would not meaningfully affect our discussion.

4 Modularly Metaprogrammable Textual Syntax

Verse, like most major contemporary programming languages, specifies a textual concrete syntax. We have chosen to specify a layout-sensitive textual concrete syntax (i.e. newlines and indentation are not ignored). This design choice is not fundamental to our proposed contributions, but it will be useful for cleanly expressing a class of examples that we will discuss later. We will specify key aspects of its concrete syntax with an *Adams grammar* [2] (such a specification for Wyvern, which has a very similar syntax, can be found in [24]), but for the purposes of this proposal, we will simply introduce Verse's concrete syntax by example as we go on.

A programming language's concrete syntax serves as its human-facing user interface, so it is common practice to build in derived forms (colloquially, *syntactic sugar*) that capture common idioms more concisely or naturally. For example, derived list syntax is built in to most dialects of ML, so that instead of having to write `Cons(1, Cons(2, Cons(3, Nil)))`, a programmer can equivalently write `[1, 2, 3]`. Many languages go further, building in syntax associated with various other types of data, like vectors (SML/NJ), arrays (Ocaml), commands (Haskell), syntax trees (Scala), XML data (Scala, Ur/Web) and SQL queries (F#, Ur/Web).

Rather than privileging particular library constructs with primitive syntactic support, Verse exposes primitives that allow library providers to introduce new derived syntax on their own. To motivate our desire for this level of syntactic control, we begin in Sec. 4.1 with a representative example: regular expression patterns expressed using abstract data types. We show how the usual workaround of using strings to introduce patterns is not ideal. We then survey existing alternatives in Sec. 4.2, finding that they involve an unacceptable loss of modularity and other undesirable trade-offs. In Sec. 4.3, we outline our proposed mechanisms and discuss how they resolve these issues. We conclude in Sec. 4.4 with a timeline for remaining work.

4.1 Motivating Example: Regular Expression Syntax

Let us begin by taking the perspective of a *regular expression* library provider. Recall that regular expressions are a common way to capture patterns in strings [33]. We will assume that the abstract syntax of patterns, p, over strings, s, is specified as below:

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

The most direct way to express this abstract syntax is by defining a recursive sum type [14]. Verse supports these as case types, which are analagous to datatypes in ML (we will see how the type structure of case types can in fact be expressed in libraries later):

```
casetype Pattern
  Empty
  Str of string
  Seq of Pattern * Pattern
  Or of Pattern * Pattern
  Star of Pattern
  Group of Pattern
```

However, there are some reasons not to expose this representation of patterns directly to clients. First, patterns are usually identified up to their reduction to a normal form. For example, $\textbf{seq}(\textbf{empty}, p)$ has normal form p. 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 patterns (e.g. a corresponding finite automata) without exposing this "implementation detail" to clients. Indeed, there are many ways to represent regular expression patterns, each with different performance trade-offs. For these reasons, a better approach is to use *abstract data types*. In Verse, as in ML, this is a mode of use of the module system. In particular, we can define the following *module signature*, where the type of patterns, `t`, is held abstract. The client of any module `P : PATTERN` can then identify patterns as terms of type `P.t`.

```
signature PATTERN
  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 : (
    'a ->
    (string -> 'a) ->
    (t * t -> 'a) ->
    (t * t -> 'a) ->
    (t -> 'a) ->
    (t -> 'a) ->
    'a)
```

By holding the representation type of patterns abstract, the burden of proving that the case analysis function above cannot be used to distinguish patterns with the same normal form is localized to each module implementing this signature. The details are standard and not particularly interesting given our purposes, so we omit them.

Concrete Syntax The abstract syntax of patterns is too verbose to be used directly in all but the most trivial examples, so patterns are conventionally written using a more concise concrete syntax. For example, the concrete syntax `A|T|G|C` corresponds to abstract syntax with the following much more verbose expression:

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

To express the concrete syntax of patterns, regular expression library providers usually provide a utility function that transforms strings to patterns. As just mentioned, there may be many implementations of the `PATTERN` signature, so the standard approach is to define a *parameterized module* (a.k.a. *functor* in SML) defining utility functions generically:

```
module PatternUtil(P : PATTERN)
  fun parse(s : string) : P.t
    (* ... pattern parser here ... *)
```

This allows the client of any module `P : PATTERN` to use the following definitions:

```
module PU = PatternUtil(P)
let pattern = PU.parse
```

to construct patterns like this:

```
pattern "A|T|G|C"
```

This approach is imperfect for several reasons:

1. Our first problem is syntactic: string escape sequences conflict syntactically with pattern escape sequences. For example, the following will not be well-formed:

   ```
   let ssn = pattern "\d\d\d-\d\d-\d\d\d\d"
   ```

   When compiling an analagous expression using `javac`, we encounter the error message `error: illegal escape character`.[4] In a small lab study, we observed that this error was common, and nearly always initially misinterpreted by even experienced programmers who hadn't used regular expressions recently [26]. The inelegant workaround is to double backslashes:

   ```
   let ssn = pattern "\\d\\d\\d-\\d\\d-\\d\\d\\d\\d"
   ```

   Some languages, anticipating such modes of use, build in special string forms that leave escape sequences uninterpretted. For example, Ocaml allows the following, which is perhaps more acceptable:

   ```
   let ssn = pattern {rx|\d\d\d-\d\d-\d\d\d\d|rx}
   ```

2. The next problem is that pattern parsing does not occur until the pattern is evaluated at run-time. For example, the following malformed pattern will only trigger an error when this expression is evaluated during the full moon:

   ```
   case moon_phase
     Full => pattern "(GC" (* malformedness not statically detected *)
     _ => (* ... *)
   ```

   Such issues can sometimes be found via testing, but empirical data gathered from 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" [31].

   We might instead attempt to treat the well-formedness of patterns constructed from strings as a static verification condition. Statically proving that this condition holds throughout a program is wildly difficult to automate in general, because it involves reasoning about arbitrary dynamic behavior. In the example above, we must know that the variable `pattern` is equivalent to the function `PU.pattern`. Moreover, if the argument were not written literally, e.g. we had written `strconcat "(G" "C"`, we would need to be able to establish that this is equivalent to the string above. It is simple to come up with arbitrarily more complex examples that thwart any putative decision procedure, particularly for patterns that are dynamically constructed based on input to the program (further discussed below).

[4]When compiling an analagous expression using SML of New Jersey (SML/NJ), we encounter the seemingly inexplicable error message `Error: unclosed string`.

Parsing patterns at run-time also incurs a performance penalty. To avoid incurring it every time the pattern is encountered (e.g. for each item in a very large dataset), one must be careful to stage the computation appropriately, or use an appropriately tuned caching strategy.

3. The final problem is that using strings to construct patterns encourages programmers to use string concatenation inappropriately to construct patterns derived from user input. For example, consider the following function:

```
fun example_bad(name : string)
  pattern (name ^ ": \\d\\d\\d-\\d\\d-\\d\\d\\d\\d")
```

   The (unstated) intent here is to treat `name` as a sub-pattern matching only itself, but this is not the observed behavior when `name` contains special characters that mean something else in patterns. The correct code again resembles abstract syntax:

```
fun example_fixed(name : string)
  P.Seq(P.Str(name), P.Seq(pattern ": ", ssn)) (* ssn as above *)
```

   In applications that query sensitive data, failures of this variety can lead to *injection attacks*. These are observed to be both common and catastrophic from the perspective of security [1]. The attack vector above was, of course, the fault of the programmer using a flawed heuristic, so it could have been avoided with sufficient discipline. The problem is again that it is difficult to detect violations of this discipline automatically. Both functions above have the same type and behave identically at many inputs, particularly those that would be expected during typical executions of the program (i.e. alphabetic names), so testing often misses the problem. Formally proving that such problems do not arise involves reasoning about complex run-time data flows.

4.2 Existing Approaches

These difficulties with string-oriented approaches to concrete syntax are not particular to regular expression patterns, but rather arise in many similar scenarios, motivating much research on reducing the need for run-time string parsing [4]. Existing approaches take one of two approaches: *syntax extension* or *static term rewriting*.

4.2.1 Syntax Extension

One common approach is to use a system that allows users to directly extend the syntax of a language with new derived syntax, in this case for patterns.

The simplest such systems are those where each new syntactic form is described by a single rewrite rule. For example, the external language of Coq (called Gallina) includes such a system [21]. A formal account of such systems has been developed by Griffin [12]. Unfortunately, these systems are not able to express pattern syntax in the conventional manner. For example, sequences of characters can only be parsed as identifiers using these systems, rather than as characters in a pattern. Syntax extension systems based on context-free grammars like Sugar* [8], Camlp4 [18] and many others are more expressive, and would allow us to directly introduce pattern syntax into our core language's grammar.

However, this is a perilous approach because none of the systems described thusfar guarantee *syntactic composability*, i.e. as the Coq manual states: "mixing different symbolic notations in [the] same text may cause serious parsing ambiguity". If another library provider used similar syntax for a different variant of regular expressions, or for an entirely unrelated abstraction, then a client could not simultaneously use both libraries in the same scope. Properly considered, every combination of extensions introduced by one of these mechanisms creates a *de facto* derived syntactic dialect of the language.

In response to this problem, Schwerdfeger and Van Wyk have developed a modular analysis that accepts only grammar extensions that use a unique starting token and satisfy some subtle conditions on follow sets of base language non-terminals [30]. Such extensions can be used together in any combination. However, simple starting tokens like `pattern` cannot be guaranteed to be globally unique, so we would need to use a more verbose starting token like `edu_cmu_verse_rx_pattern`. There is no principled way to define local abbreviations for starting tokens because this mechanism is language-external.

Putting this issue aside, we must now decide how our newly introduced derived forms elaborate to forms in our base grammar. In this case, because we are using a modular encoding of patterns, we must first decide which module the desugaring will use. Clearly, simply assuming that a module identified as `P` satisying `PATTERN` is in scope is a brittle solution. Indeed, we should expect that the extension mechanism actively prevents such capture of specific variable names to ensure that variables (including module variables) can be freely renamed. Such a *hygiene discipline* is well-understood when performing term-to-term rewriting (discussed below) or in simple rewrite systems like those found in Coq. For more flexible mechanisms, the issue is a topic of ongoing research (none of the grammar-based mechanisms described above enforce hygiene).

Putting aside the question of hygiene as well, we can address the problem by requiring that the client explicitly identify the module the desugaring should use:

```
let ssn = edu_cmu_verse_rx_pattern P /\d\d\d-\d\d-\d\d\d\d/
```

For patterns constructed compositionally, we can define *splicing* syntax for both strings and other patterns:

```
fun example_syn(name : string)
  edu_cmu_verse_rx_pattern P /@name: %ssn/
```

The body of this function elaborates to the body of `example_fixed` as shown above. Had we mistakenly used the pattern splicing prefix, `%name`, rather than the string splicing prefix, `@name`, we would get a static type error, rather than a silent security vulnerability.

Such syntax extension approaches suffer from a paucity of direct reasoning principles. Given an unfamiliar piece of syntax, there is no simple method for determining what type it will have, or even for identifying which extension determines its desugaring, causing difficulties for both humans (related to code comprehension) and tools.

4.2.2 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*. Among the earliest examples is the LISP macro system [15]. This system was later refined, notably in Scheme and its dialects to better support reasoning compositionally about the rewriting that is performed [17]. In languages with a stronger static type discipline, variations on macros that restrict rewriting to a particular type and perform the rewriting in an explicitly staged manner have also been studied [16, 11] and integrated into languages, e.g. MacroML [11] and Scala [5].

The most immediate problem with using these in our example is that we are not aware of any such statically-typed macro system that integrates cleanly with an ML-style module system. However, let us imagine a macro system that would allow us to repurpose string syntax as follows:

```
let ssn = pattern P {rx|\d\d\d-\d\d-\d\d\d\d|rx}
```

Here, `pattern` is a macro parameterized by a module `P : PATTERN`. It statically parses the provided string literal (which must still be written using an Ocaml-style literal here) to generate an elaboration, at type `P.t`.

For patterns that are constructed compositionally, we need to get more creative. For example, we might repurpose the infix operators that the language provides for other purposes:

```
fun example_macro(name : string)
  pattern P (name ^ ": " + ssn)
```

This does not give us syntax that is quite as clean and clear as the syntax available using syntax extensions, but it does address a number of its deficiencies: there cannot be syntactic conflicts (because the syntax is not extended), we can define macro abbreviations because macros are integrated into the language, there is a hygiene discipline that guarantees that the elaboration will not capture variables inadvertently, and there is a typing discipline (i.e. we need not determine the elaboration to know what type it will have).

4.3 Contributions

We propose a new primitive, the **typed syntax macro** (TSM), that combines the flexibility of syntax extensions with the reasoning guarantees of statically-typed macros, and show how to integrate it into a language with an ML-style module system, addressing all of the problems described above. We then introduce **type-specific languages** (TSLs), which leverage local type inference to control TSM dispatch. This further decreases syntactic cost (which is, of course, the entire purpose of this exercise). The result is that derived syntax defined using TSLs approaches the cost of derived syntax built in primitively.

4.3.1 Typed Syntax Macros (TSMs)

As our first example, consider the following concrete term:

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

Here, a parameterized TSM `pattern` is applied to a module parameter, `P`, and a delimited form, `/A|T|G|C/`. Based on the body of the delimited form, i.e. the characters between the delimiters, the TSM statically computes an elaboration to the following base external form:

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

The parameterized TSM `pattern` is defined as shown below:

```
syntax pattern(P : PATTERN) at P.t
  fn (ps : ParseStream) => (* pattern parser here *)
```

We declare a module parameter (calling it `P` here is unimportant, i.e. the TSM can be used with *any* module satisfying module type `PATTERN`), which is bound in the scope of the type clause **`at`** `P.t`. This clause guarantees that all elaborations successfully generated by this TSM will be of type `P.t`. Elaborations are generated by the static action of the parse function defined in the indented block on the next line. The parse function must be of type `ParseStream -> Exp`, where the type `ParseStream` gives the function access to the *body* of the delimited form (in blue above) and the type `Exp` is a case type that encodes the abstract syntax of the base language (i.e. the Verse EL). Both types are defined in the Verse *prelude*, which is a set of definitions available ambiently.

In positions where the type is known, e.g. due to a type ascription or at the argument position of a function call, the module parameter `P` can be inferred:

```
let N : P.t = pattern /A|T|G|C/
```

TSMs can be partially applied and abbreviated using a let-binding style mechanism:

```
let syntax pat = pattern P
let ssn = pat /\d\d\d-\d\d-\d\d\d\d/
```

TSMs can treat portions of the body of the delimited form as a spliced base language term, to support splicing syntax:

```
fun example_tsm(name: string)
  pat /@name: %ssn/
```

A hygiene mechanism ensures that only those portions of the elaboration derived from such spliced terms can refer to variables in the surrounding scope, preventing inadvertent variable capture by the elaboration. In other words, the generated elaboration must be context-independent.

Formalization To give a flavor of the formal specification underlying TSMs, let us look at the rule for handling TSM invocation in a synthetic position.

$$\frac{\Delta \vdash s \mathbin{@} \sigma\ \{\iota_{\text{parser}}\} \qquad \mathbf{parsestream}(\mathit{body}) = \iota_{\text{ps}} \qquad \iota_{\text{parser}}(\iota_{\text{ps}}) \Downarrow \iota_{\text{elab}} \qquad \iota_{\text{elab}} \uparrow e_{\text{elab}} \qquad \Delta; \Gamma; \emptyset; \emptyset \vdash e_{\text{elab}} \Leftarrow \sigma \leadsto \iota}{\Delta_{\text{out}}; \Gamma_{\text{out}}; \Delta; \Gamma \vdash \mathbf{invoketsm}(s; \mathit{body}) \Rightarrow \sigma \leadsto \iota}\ \text{(syn-aptsm)}$$

Note that the judgement forms are slightly simplified in that they omit various contexts relevant only to other Verse primitives. We include only *typing contexts*, Γ, which map variables to types, and *type abstraction contexts*, Δ, which track universally quantified type variables (both in the standard way). The premises have the following purposes:

1. The first premise can be pronounced "Under Δ, TSM expression s is a TSM at type σ with parser ι_{parser}". Application of a parameterized TSM to its parameters is handled by this judgement (not shown).
2. The second premise creates a parse stream, ι_{ps}, from the body of the delimited form.
3. The third premise applies the parser to the parse stream to generate an encoding of the elaboration, ι_{elab}.
4. The fourth premise decodes ι_{elab}, producing the elaboration e_{elab}.
5. The fifth premise validates the elaboration by analyzing it against the type σ under an empty context. The *current* typing and kinding contexts are "saved" for use when a spliced term is encountered during this process by setting them as the new *outer contexts*.

   Variables in the outer contexts cannot be used directly by the elaboration, e.g. they are ignored by the (syn-var) rule:

$$\frac{}{\Delta_{\text{out}}; \Gamma_{\text{out}}; \Delta; \Gamma, x : \sigma \vdash x \Rightarrow \sigma \leadsto x}\ \text{(syn-var)}$$

   The outer contexts are switched back in only when encountering a spliced expression, which is marked:

$$\frac{\emptyset; \emptyset; \Delta_{\text{out}}; \Gamma_{\text{out}} \vdash e \Rightarrow \sigma \leadsto \iota}{\Delta_{\text{out}}; \Gamma_{\text{out}}; \Delta; \Gamma \vdash \mathbf{spliced}(e) \Rightarrow \sigma \leadsto \iota}\ \text{(syn-spliced)} \qquad \frac{\emptyset; \emptyset; \Delta_{\text{out}}; \Gamma_{\text{out}} \vdash e \Leftarrow \sigma \leadsto \iota}{\Delta_{\text{out}}; \Gamma_{\text{out}}; \Delta; \Gamma \vdash \mathbf{spliced}(e) \Leftarrow \sigma \leadsto \iota}\ \text{(ana-spliced)}$$

   For example, the elaboration generated in `example_tsm` above would be:

```
P.Seq(P.Str(spliced(name)), P.Seq(P.Str ": ", spliced(ssn)))
```

We will give a more complete account of all of these judgements later.

4.3.2 Type-Specific Languages (TSLs)

To further lower the syntactic cost of using TSMs, Verse also supports *type-specific languages* (TSLs), which allow library providers to associate a TSM directly with a declared type. For example, a module `P` can associate `pattern` with `P.t` as follows:

```
module P : PATTERN with syntax pattern at t
  type t = (* ... *)
  (* ... *)
```

Local type inference then determines which TSM is applied when analyzing a delimited form not prefixed by a TSM name. For example, the following is equivalent to the above:

```
fun example_tsl(name : string) : P.t
  /@name: %ssn/
```

As another example, we can use TSLs to express derived list syntax. For example, if we use a case type, we can declare the TSL directly upon declaration as follows:

```
casetype list('a) {
  Nil
  Cons of 'a * list('a)
} with syntax
  fn (body : ParseStream) => (* ... comma-delimited spliced exps ... *)
```

Together, this allows us to write a list of patterns like this:

```
let x : list(P.t) = [/\d/, /\d\d/, /\d\d\d/]
```

4.4 Timeline

We have described and formally specified TSMs in a recently published paper [25], and TSLs in a paper last year [24], both in the context of the Wyvern language. In the context of Verse, there are some changes that need to be made in the formalization. In particular, neither paper covered integration with an ML-style module system. We plan to complete this in the course of writing the dissertation.

5 Modularly Metaprogrammable Type Structure

The Verse EL also does not build in substantial type structure. Only functions are built in directly. The type structure of case types (shown in use above), tuples, records, objects and other more esoteric constructs that we will describe later can all be expressed as modes of use of *metamodules* (which are distinct from, though conceptually related to, *modules*).

The remainder of this section is organized like Section 4. We begin with an example of type structure that cannot be expressed in languages like ML directly in Sec. 5.1, then show how existing approaches are insufficient in Sec. 5.2. Then we outline our approach in Sec. **??** and conclude with a timeline for remaining work in Sec. **??**.

5.1 Motivating Example: Labeled Products and Regular Strings

As a simple introductory example, let us consider *labeled products* and *regular strings*. In Verse, a simple labeled product type classifying conference papers can be defined like this:

```
let type Paper = lprodtype {
  title : rstring /.+/
   conf : rstring /([A-Z]+) (\d\d\d\d)/
}
```

The first row is labeled `title` and specifies the regular string type `rstring /.*/`. Regular string types classify values that behave as strings known to be in the regular language specified by the regular expression given as the type parameter. Here, we have that paper titles must be non-empty strings. The second row is labeled `conf` and specifies a regular string type with two parenthesized groups, corresponding to the conference abbreviation and the year. The **`let type`** construction defines a type synonym, `Paper`. Alternatively, we could have used the following construction to declare `Paper` *generatively*:

```
lprodtype Paper {
  title : rstring /.+/
   conf : rstring /([A-Z]+) (\d\d\d\d)/
}
```

The difference is in how type equality is handled. Two labeled product types with the same signature are equal, unless declared generatively, in which case type equality is nominal.

In either case, we can introduce a value of type `Paper` in an analytic position in one of two ways. We can omit the labels:

```
let exmpl : Paper = {"An Example Paper", "EXMPL 2015"}
```

Alternatively, we can include them explicitly for clarity:

```
let exmpl : Paper = {title => "An Example Paper", conf => "EXMPL 2015"}
```

We can then project a row out of such a value using the row projection operator:

```
let exmpl_conf = # ‘conf exmpl
```

Here, `#` is an operator parameterized by a static label, written literally `‘conf`, and taking one argument, `exmpl`.

We can also project captured groups out of a regular string using the `#group` operator, which is parameterized by a group index:

```
let exmpl_conf_name = #group 0 exmpl_conf
```

The variable `exmpl_conf_name` has type `rstring /[A-Z]+/`.

Both labeled products and regular strings support a number of other operations. For example, labeled products can be concatenated (with any common rows updated with the value on the right):

```
let a : lprodtype {authors : list(rstring /.+/)} = {["Harry Q. Bovik"]}
let exmpl_paper_final = lprod+ exmpl_paper a
```

We can drop a row as well:

```
let exmpl_paper_anon = lprod- exmpl_paper_final ‘authors
```

5.2 Existing Approaches

5.2.1 Records and Tuples

Recall that Verse builds in only nullary and binary products into its internal language, so the type structure described above for labeled products is not directly expressible using IL primitives. If we weaken slightly our insistence on minimalism and built in record/tuple types, as in other functional languages, we have options that get close to (but do not exactly achieve) this type structure. In particular, unlike labeled product types, record types are identified up to row reordering. Using a record with the same row labels would prevent us from omitting the labels and relying on positional information when introducing a value of record type in an analytic position, as we did with type `Paper` above.

One workaround is to replace each labeled product type in the program with a record type with row labels tagged by position (recovering the type disequality we seek to express with labeled product types) and conversion functions from the corresponding unlabeled product type and untagged record type to this type. This partially recovers some of the syntactic cost we seek, but introduces much additional boilerplate as well as run-time cost. We are interested in such issues, so this is not an entirely satisfying solution. Moreover, this still does not give us support for the extra operations, like concatenation, shown above.

5.2.2 Refinement Types

For regular strings, one could instead use standard strings (which themselves may be lists or vectors of characters together with derived syntax), inserting run-time checks around introductory forms and into other operations (with one exception, discussed below). This would, of course, weaken the static semantics and add both syntactic and run-time cost. To recover the static guarantees of regular strings, we could treat membership of a string in a specified regular language as a static verification condition, moving the logic governing regular string types out of the language and into an external *refinement type system* [10].

One problem with this approach is that there is no direct analog to the group projection operator described above. The static and dynamic semantics of this operator depend on information available only in the static type of the regular string (i.e. the position of the captured group). Refinement types are strictly a layer above the language, and thus cannot affect its dynamic semantics.

A related problem is that there is no feasible way to use refinement types to introduce type disequalities that can later affect the internal representation of a value. All standard strings are represented uniformly. The fact that some value of type `string` can be given a refined type `rstring /A|B/` cannot be justification to locally alter its representation (e.g. to a single bit in this case). For regular strings, there may be limited benefit to doing this, but in general, this can have significant performance ramifications. For example, it is common to justify the omission of a primitive type of “unsigned integers representable in 32 bits” by the fact that this is “merely” a refinement of arbitrary precision integers, but in so doing one loses the ability to use a machine representation more suitable to this invariant. This is also the reason why one cannot viably do away with primitive record types by claiming that they are refinements of finite mappings.

Put another way, refinement types are useful when one only needs to make finer static distinctions within an existing type. When one needs to introduce new operators or type disequalities that have “downstream” impact, refinement types are not the right solution.

5.3 Contributions

An important point is that labeled products, as well as other “record-like” constructs, can be expressed by a *type-directed translation* targeting our IL (and indeed, many compilers take advantage of facts like this to simplify the number of cases that must be considered downstream). Substantially more sophisticated examples require at most only a very slightly more capable intermediate language (the IL of popular virtual machines, even those with flaws like ubiquitous `null` references, e.g. JVM or CIL bytecode, are also sufficient if used strictly as translation targets).

Metamodules introduce type and operator constructors parameterized by arbitrary *static values* and metaprogrammatically express the logic governing their typechecking and translation to the IL using the Verse *static language* (SL). Metamodules are classified by *metasignatures*. For example, let us consider the following metasignature:

```
metasignature LPROD
  tycon c of LblTyList
  ana opcon intro_unlabeled
  ana opcon intro_labeled of LblList
  syn opcon # of Lbl
```

reworking active tycons as metasignatures and metastructures is compelling, but I need to figure it out. will continue writing.

References

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

[2] M. D. Adams. Principled Parsing for Indentation-Sensitive Languages: Revisiting Landin's Offside Rule. In *POPL 2013*, pages 511–522, New York, NY, USA, 2013. ACM.

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

[4] M. Bravenboer, E. Dolstra, and E. Visser. Preventing Injection Attacks with Syntax Embeddings. In *GPCE '07*, pages 3–12, New York, NY, USA. ACM.

[5] 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, New York, NY, USA, 2013. ACM.

[6] A. Chlipala. Ur/web: A simple model for programming the web. In S. K. Rajamani and D. Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 153–165. ACM, 2015.

[7] S. Erdweg, T. Rendel, C. Kastner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *OOPSLA '11*.

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

[9] M. Fluet, M. Rainey, J. H. Reppy, A. Shaw, and Y. Xiao. Manticore: a heterogeneous parallel language. In N. Glew and G. E. Blelloch, editors, *Proceedings of the POPL 2007 Workshop on Declarative Aspects of Multicore Programming, DAMP 2007, Nice, France, January 16, 2007*, pages 37–44. ACM, 2007.

[10] T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation*, pages 268–277, Toronto, Ontario, June 1991. ACM Press.

[11] S. E. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: type-safe, generative, binding macros in macroml. In *ACM SIGPLAN Notices*, volume 36, pages 74–85. ACM, 2001.

[12] T. Griffin. Notational definition-a formal account. In *Logic in Computer Science, 1988. LICS '88., Proceedings of the Third Annual Symposium on*, pages 372–383, 1988.

[13] R. Harper. Programming in Standard ML, 1997.

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

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

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

[17] 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. To appear in Lisp and Symbolic Computation.

[18] 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.

[19] D. MacQueen. Modules for standard ml. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 198–207, New York, NY, USA, 1984. ACM.

[20] L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *PPDP*, pages 82–93. ACM, 2005.

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

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

[23] T. Murphy, VII., K. Crary, and R. Harper. Type-safe Distributed Programming with ML5. In *Proceedings of the 3rd Conference on Trustworthy Global Computing*, TGC'07, pages 108–123, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[25] C. Omar, C. Wang, and J. Aldrich. Composable and hygienic typed syntax macros. In *ACM Symposium on Applied Computing*, 2015.

[26] 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 2012, pages 859–869, Piscataway, NJ, USA, 2012. IEEE Press.

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

[28] 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.

[29] A. Rossberg, D. L. Botlan, G. Tack, T. Brunklaus, and G. Smolka. *Alice Through the Looking Glass*, volume 5 of *Trends in Functional Programming*, pages 79–96. Intellect Books, Bristol, UK, ISBN 1-84150144-1, Munich, Germany, Feb. 2006.

[30] A. Schwerdfeger and E. V. Wyk. Verifiable composition of deterministic grammars. In M. Hind and A. Diwan, editors, *PLDI*, pages 199–210. ACM, 2009.

[31] 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*, pages 20–26. ACM, 2012.

[32] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *PLDI '96*, Philadelphia, PA, May 1996.

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

[34] Xi, Chen, and Chen. Guarded recursive datatype constructors. In *POPL: 30th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2003.