Modularly Programmable Syntax and Type Structure (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 and type structure. For example, nearly all major functional languages include derived forms for lists, but introducing derived forms for other library constructs requires forming new dialects of these languages. Programmers have no way to modularly combine such 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 dialects by giving library providers the ability to programmatically control syntactic expansion, typechecking and translation (to a small typed internal language) in a safe and modular manner. We validate the expressiveness of these primitives by giving a number of examples of library constructs that, in other major languages, are or would need to be built in primitively.

1 Motivation

Functional programming language designers have long studied small typed lambda calculi to develop a principled understanding of fundamental metatheoretic 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 typically feature various generalizations and primitive "embellishments" motivated by a consideration of human factors. For example, major languages like Standard ML (SML) [36, 24], OCaml [32] and Haskell [30] 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 useful libraries. However, a stable language design that fully achieves this ideal has yet to emerge, as evidenced by the diverse array of "dialects" 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.

[1] Throughout this work, phrases that should be read as having an intuitive meaning, rather than a strict mathematical meaning, will be first 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 [16].

1.1 Why are there so many dialects?

This calls for an investigation: why is it that programmers and researchers are still so often unable to satisfyingly express the constructs that they need in libraries, as modes of use of the "general-purpose" primitives already available in major languages today, and instead see a need for new language dialects?

Perhaps the most common motivation may be that the *syntactic cost* of expressing a construct of interest using contemporary general-purpose primitives is not always ideal. 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 [8]) that builds in derived forms for SQL queries, HTML elements and other datatypes used in the domain of web programming [9]. These are certainly not the only types of data that could similarly benefit from the availability of specialized derived forms – we will consider another example in Sec. 4.1. Tools like Camlp4 [32], Sugar* [11, 12] and Racket [14], which we will discuss in Sec. 4.2, have lowered the engineering costs of constructing syntactic dialects in such situations, contributing to their proliferation.

More advanced dialects introduce new type structure, going beyond what is possible with only new derived forms. As a simple example, the static and dynamic semantics of records cannot be expressed by context-independent expansion to a language with only nullary and binary products. Various languages have explored "record-like" primitives that go further, supporting functional update operators, width and depth coercions (sometimes implicit), methods, prototypic dispatch and other such "semantic embellishments" that in turn cannot be expressed by context-independent expansion to a language with only standard record types (we will detail an example in Sec. 5.1). OCaml primitively builds in the type structure of polymorphic variants, open datatypes and operations that use format strings like `sprintf` [32]. ReactiveML builds in primitives for functional reactive programming [34]. ML5 builds in high-level primitives for distributed programming based on a modal lambda calculus [37]. Manticore [15] and AliceML [43] build in parallel programming primitives with a more elaborate type structure than is found in simpler accounts of parallelism. MLj builds in the type structure of the Java object system (motivated by a desire to interface safely and naturally with Java libraries) [4]. Other dialects do the same for other foreign languages, e.g. Furr and Foster describe a dialect of OCaml that builds in the type structure of C [19]. Tools like proof assistants and logical frameworks are used to specify and reason metatheoretically about dialects like these, and tools like compiler generators and language frameworks [13] lower their implementation cost, again contributing to their proliferation.

1.2 Dialects Considered Harmful

Some have viewed the ongoing proliferation of dialects described above as harmless or even as desirable, arguing that programmers can simply choose the right language for the job [51]. 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 in another dialect. As the study of MLj demonstrated, addressing this interoperability problem requires somehow "combining" the dialects into a single language. 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 syntactic and semantic 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, as suggested by the diversity of examples given above, 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 [38]). 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 to be flawed in some way cannot be removed or changed without breaking backwards compatibility.

This suggests that language designers should be quite careful and strive to keep general-purpose languages small and free of *ad hoc* primitives. Given that we also want to avoid dialect formation, for the reasons described above, this leaves two possible paths forward. One, exemplified (arguably) by SML, is to simply eschew further syntactic conveniences and innovative type structure 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 languages 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 [52]. 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 (note, however, that 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. In particular, we plan to introduce the following primtives:

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). We begin by showing explicitly-invoked TSMs, then show how to associate a privileged TSM with a type declaration and then rely on a local type inference scheme to invoke these implicitly.

2. **Metamodules**, introduced in Sec. 5, reduce the need to primitively build in the type structure of constructs like records (and variants thereof), labeled sums and other interesting constructs that we will introduce later by giving library providers programmatic "hooks" directly into the semantics, which are specified as a *type-directed translation semantics* targeting a small *typed internal language* (introduced in Sec. 3).

Both TSMs and metamodules make extensive use of *static code generation* (also called *static* or *compile-time metaprogramming*, hence the name *metamodules*), meaning that the relevant rules in the static semantics of the language call for the evaluation of *static functions* that generate static representations of expressions and types. 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 and compilation process (but did not take a type theoretic approach) [50].

As vehicles for this work, we plan to formally specify small typed lambda calculi that capture each of the novel primitives that we introduce "minimally". We will also describe (but not formally specify) a new "full-scale" functional language called Verse.[3] The reason we will not follow Standard ML [36] 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. We will give a brief overview covering how these languages are organized in Chap. 3.

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 metamodules to maintain:

- a *hygienic type discipline*, meaning that these constructs maintain type safety and that the validity of the code generated will not make any assumptions about 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]

We will make these notions completely precise as we continue.

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 and new types and operators atop a small type-theoretic internal language 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 dialects.

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

[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.

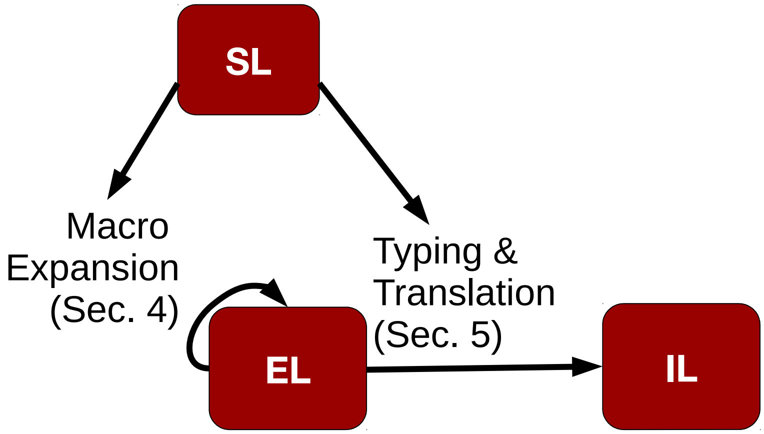

Figure 1: A diagrammatic overview of the core language.

be sensible to use the mechanisms we introduce (following the example of languages that support operator overloading or type classes [22], which also have the potential for such "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 [35]) or to the program extraction mechanisms of proof assistants like Coq with modifications, but do not plan to pursue this line of research in this dissertation.

3 Language Overview

Let us begin with a brief overview of how Verse and the minimal calculi that we will develop are organized and specified at a high level. We will assume that readers are familiar with the early chapters of a textbook like *TAPL* [41] or *PFPL* [25].

3.1 Core Language

The Verse core language – the language of types and expressions – will be the focus of our efforts. Figure 1 gives a diagrammatic overview of how the core language is organized. The key novelty is that the core language is split into a user-facing *typed external language* (EL) specified by type-directed translation to a minimal *typed internal language* (IL). Notionally, one might think of this design as shifting the first stage of a type-directed compiler (e.g. the TIL compiler for Standard ML [48]) "one level up" into the language itself. A third sublanguage, the *static language* (SL), is involved in giving library providers programmatic control over aspects of both macro expansion (which involves generating external terms) and this type-directed translation from the EL to the IL (which involves generating internal terms). We summarize the main judgements relevant to the IL, EL and SL below. Readers who prefer to see examples first can skim these sections for now, returning as needed.

3.1.1 Internal Language

Programs ultimately evaluate as *internal expressions*, ι. Internal expressions are classified by *internal types*, τ, so the internal language forms a standard typed lambda calculus. For our purposes, it suffices to use the strictly evaluated polymorphic lambda calculus with nullary and binary product and sum types and recursive types as our IL. We assume in this proposal that the reader is familiar with this language (we follow *PFPL* [25] directly). The main judgements in the static semantics of the IL take the following familiar form (omitting contexts for simplicity throughout this section):

Judgement Form	**Pronunciation**
$\vdash \tau\ \mathtt{itype}$	Internal type τ is valid.
$\vdash \iota : \tau$	Internal expression ι is assigned internal type τ.

The dynamic semantics are specified also in the standard manner as a transition system with judgements of the following form:

Judgement Form	**Pronunciation**
$\iota \mapsto \iota'$	Internal expression ι transitions to ι'.
$\iota\ \mathtt{val}$	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 $\iota'\ \mathtt{val}$.

Note that features like state, exceptions, basic concurrency primitives, scalars, arrays, I/O primitives and others characteristic of a first-stage compiler intermediate language would also be included in the IL in practice, and for several of these, this would affect the shape of the internal semantics. However, our design is largely insensitive to such details – the only strict requirements are that the IL be type safe and support parametric type abstraction. To give a simple account of our novel contributions, we will stick to this small, familiar IL for the purposes of this work.

3.1.2 External Language

The external language (EL) supports a richer syntax and type structure atop the IL. Most programmers will interact exclusively with the EL, so throughout this work, the words "expression" and "type" used without qualification refer to external expressions and types, respectively. The central judgements in the specification of the EL take the following form (again omitting various contexts, which we will introduce progressively):

Judgement Form	**Pronunciation**
$\vdash \sigma\ \mathtt{type} \leadsto \tau$	Static expression σ is a type with translation τ.
$\vdash e \Rightarrow \sigma \leadsto \iota$	Expression e synthesizes type σ and has translation ι.
$\vdash e \Leftarrow \sigma \leadsto \iota$	Expression e analyzes against type σ and has translation ι.

The main points to note here are that:

- External types are distinct from internal types (thus distinguishing this style of specification from the Harper-Stone elaboration semantics for Standard ML [26]).
- The expression typing judgements are *bidirectional*, i.e. we make a judgemental distinction between *type synthesis* (the type is an "output") and *type analysis* (the type is an "input") [42]. This will allow us to explicitly specify how Verse's *local type inference* works, and in particular, how it interacts with the primitives that we aim to introduce. Like Scala, we do not seek to support non-local type inference.
- The dynamic behavior of an external expression is determined directly by its translation to the IL, i.e. there is no separate transition system governing the dynamics of external expressions.

> Proving type safety amounts to proving that the translation of every external expression of type σ is an internal expression of type τ, where τ is the translation of σ (and thus, by type safety of the IL, evaluation does not "go wrong"). This corresponds notionally to the idea of *type-preserving compilation* in type-directed compilers [48].

3.1.3 Static Language

Finally, the static language plays an important role in the semantics of the novel primitives that are the topic of this proposal. The static language is simply another typed lambda calculus consisting of *static expressions*, σ, classified by *sorts*[6], π, according to the following judgements (the "static statics"):

Judgement Form	**Pronunciation**
$\vdash \pi$ `sort`	Sort π is valid.
$\vdash \sigma :: \pi$	Static expression σ has sort π.

The "static dynamics" are specified as a transition system with judgements of the following basic form (though again omitting some necessary contexts):

Judgement Form	**Pronunciation**
$\sigma \mapsto \sigma'$	Static expression σ transitions to σ'.
σ `sval`	Static expression σ is a static value.
σ `styerr`	Static expression σ raises an external type error.

The iterated static transition judgement $\sigma \mapsto^* \sigma'$ is the reflexive, transitive closure of the static transition judgement, and the static evaluation judgement $\sigma \Downarrow \sigma'$ is derivable iff $\sigma \mapsto^* \sigma'$ and σ' `sval`.

As suggested by the form of the external type translation judgement above, external types are static expressions. More specifically, external types are static values of a primitive sort Ty. We will return to how this works in Sec. 5.

3.2 Module Language

The Verse module system is taken directly (up to small syntactic changes) from Standard ML, with which we assume a working familiarity for the purposes of this proposal [24, 33] (a related module system, e.g. OCaml's, would work just as well for our purposes). We will give examples of its use in Sec. 4 and Sec. 5.

Formally, our approach will be to first specify our contributions assuming a language without an ML-style module system, then make modifications to this specification to make it compatible with the presence of a module system. Because it has been thoroughly studied in the literature, we will defer to prior work both here and in the dissertation for many formal details. In particular, it will suffice to assume that modules are being tracked by a suitable *module context*, without detailing how this context is populated.

4 Modularly Programmable Textual Syntax

To begin, let us assume a standard, ML-like semantics for both the EL and SL and consider the EL's concrete syntax. Verse, like most major contemporary languages, specifies a textual concrete syntax.[7] 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

[6] We use the word "sort" to avoid confusion with the phrase "static type". Our use of the word "sort" should be considered distinct from the various other uses of this word found in the literature on programming languages.

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

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. 4.1 by detailing another example for which such a mechanism would be useful: regular expression patterns expressed using abstract data types. In Sec. 4.2, we then demonstrate that the usual approach of using dynamic string parsing to introduce 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 Sec. 4.3, we outline our proposed alternatives – *typed syntax macros* (TSMs) and the related *type-specific languages* (TSLs) – and discuss how they resolve these issues. We also give an overview of how TSMs are specified in terms of the judgement forms that were introduced in the previous section. We conclude in Sec. 4.4 with a concrete timeline for the remaining work.

4.1 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 [49]. 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, p, over strings, s, is specified as below:

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

One way to express this abstract syntax is by defining a recursive sum type [25]. Verse supports these as datatypes, which are comparable to datatypes in ML (we plan to show how the type structure of datatypes can in fact be expressed in libraries later):

```
datatype 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, regular expression 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 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 generative type abstraction. In particular, we can define the following *module signature*, where the type of patterns, `t`, is held abstract:

```
signature PATTERN = 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 ->
      'a -> (* Empty case *)
      (string -> 'a) -> (* Str case *)
      (t * t -> 'a) -> (* Seq case *)
      (t * t -> 'a) -> (* Or case *)
      (t -> 'a) -> (* Star case *)
      (t -> 'a) -> (* Group case *)
      'a)
  }
```

Clients of any module `P` that has been opaquely ascribed `PATTERN`, written `P :> PATTERN`, manipulate patterns as values of the type `P.t` using the interface described by this signature. The identity of this type 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:

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

4.2 Existing Approaches

4.2.1 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 `PATTERN` signature, the usual approach is to define a parameterized module (a.k.a. a *functor* in SML) defining utility functions like this abstractly:

```
module PatternUtil(P : PATTERN) => mod {
  fun parse(s : string) : P.t => (* ... pattern parser here ... *)
}
```

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

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

to construct patterns like this:

```
pattern "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 = pattern "\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.`[8] In a small lab study, we observed that this class of error was common, and often confused even experienced programmers if they had not used regular expressions recently [40]. The standard workaround has higher syntactic cost – we must double all backslashes:

[8] 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`.

```
   let val ssn = pattern "\\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 = pattern {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 this function from strings to patterns:

   ```
   fun example(name : string) =>
     P.Seq(P.Str(name), P.Seq(pattern ": ", 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 use equivalently use splicing syntax:

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

   Here, 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) =>
     pattern (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 those that would be expected during typical executions of the program (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 => pattern "(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" [46].

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 `pattern` is equal to the parse function `PU.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, the problem is not solved by evaluating the expression statically (or, more generally, in some earlier "stage" of evaluation) [29].

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

In a language that primitively supported derived pattern syntax, the 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 => rx_match (pattern "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).[9] 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 a language that primitively supported derived pattern syntax, the expansion is computed at compile-time and so there is 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. 4.2.2 and 4.2.3 respectively.

4.2.2 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 [35]. A formal account of such a system has been developed by Griffin [21]. Unfortunately, a single equation is not enough to allow us to express pattern syntax following the usual conventions. For example, there is no way to handle sequences of characters (they can only be treated as variables) or escape characters (there is no way to define disjunctive syntax) using a system like Coq's.

Other syntax extension systems are more flexible. For example, many are based on context-free grammars, e.g. Sugar* [12], Camlp4 [32] and several others. Other systems give library providers direct programmatic access to the parse stream, like Common Lisp's

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

reader macros [47] (which are distinct from its term-rewriting macros, described below) and Racket's preprocessor [14]. All of these would allow us to add pattern syntax into our core language's grammar, perhaps following Unix conventions like this:

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

As discussed above, we can define *splicing syntax* that allows us to write string and pattern expressions inline (distinguished by prefixes `@` and `%`, respectively). For example, we can write `example` equivalently, but at lower syntactic cost (cf. `example_shorter`).

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

In response to this problem, Schwerdfeger and Van Wyk developed a modular analysis that accepts only grammar extensions that specify a unique starting token and obey subtle constraints on the follow sets of base language non-terminals [44]. Extensions that satisfy these criteria can be used together in any combination without the possibility of syntactic conflict. However, the most natural starting tokens like `pattern` 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 ecosystem. This forces us to use a more verbose token like `edu_cmu_verse_rx_pattern`. 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 syntactic forms.

Putting this aside, we must also consider the question of how newly introduced forms expand to forms in our base grammar. In our example, this is tricky because we have defined a modular encoding of patterns – which particular module should the expansion use? Clearly, simply assuming that some module identified as `P` matching `PATTERN` 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 (including 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, this issue has not been adequately addressed. The library provider must simply be careful not to make assumptions about variable names and instead require that the client explicitly identify the module they intend to use as an "argument" in the new form:

```
let val ssn = edu_cmu_verse_rx_pattern P /\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.

4.2.3 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 [27] 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 [31]. In languages with a stronger static type discipline, variants of macros that restrict rewriting to a particular type and perform the rewriting statically have also been studied [28, 20] and integrated into languages, e.g. MacroML [20] 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, i.e. 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 = pattern P {rx|\d\d\d-\d\d-\d\d\d\d|rx}
```

The definition of `pattern` might look like this:

```
macro pattern[Q : PATTERN](e) at Q.t {
  static fun f(e :: Exp) :: Exp => case(e) {
      StrLit(s) => (* pattern parser here *)
    | BinOp(Caret, e1, e2) => ‘Q.Seq(Q.Str(%e1), %(f e2))‘
    | BinOp(Plus, e1, e2) => ‘Q.Seq(%(f e1), %(f e2))‘
    | _ => raise Error
  }
}
```

Here, `pattern` is a macro parameterized by a module satisfying the signature `PATTERN` (we identify it as `Q` to emphasize that there is nothing special about the identifier `P` that we have been using) and taking a single argument, identified as `e`. The macro specifies that the expansion it statically generates will be of type `Q.t`. It is defined by a static function that examines the syntax tree of the provided argument (syntax trees are of a sort `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 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 as follows:

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

The logic implementing this can be seen above. We assume that there is derived syntax available for the sort `Exp` where the `%` prefix indicates a spliced expression, i.e. *quasiquotation* as in Lisp [3] and Scala [45].

Having to creatively repurpose existing syntax in this way limits the effect a library provider can have on syntactic cost (particularly when it would be desirable to adopt conventions that are not consistent with 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 still 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 a macro application must have.

4.3 Contributions

Verse provides a new external primitive – the **typed syntax macro** (TSM) – that combines the syntactic flexibility of syntax extensions with the reasoning guarantees of typed macros. TSMs can be parameterized by modules, so they can be used to define syntax valid at any type defined by a module satisfying a specified signature. As we will discuss in Sec. 4.3.1 below, this addresses all of the problems brought up above, at moderate syntactic cost.

To further decrease syntactic cost (which is, of course, the entire purpose of this effort), we go on in Sec. 4.3.2 to briefly introduce **type-specific languages** (TSLs). TSLs are TSMs associated directly with a type when it is generated. Verse uses local type inference to implicitly control TSL dispatch.

Syntax defined by library providers using TSMs and TSLs comes at syntactic cost comparable to that of derived syntax built in primitively by the language designer or using a naïve syntax extension tool like Camlp4, without causing dialect formation. As such, these *ad hoc* approaches can, in large part, be discarded. That said, we examine some limitations of our approach in Sec. 4.3.3.

4.3.1 Typed Syntax Macros (TSMs)

To introduce TSMs, let us consider the following concrete external expression:

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

Here, we apply a *parameterized TSM*, `pattern`, first to a module parameter, `P`, then to a *delimited form*, `/A|T|G|C/`. Note that a number of alternative delimiters are also provided by Verse's concrete syntax and could equivalently be used. The TSM statically parses the *body* of the provided delimited form, i.e. the characters between the delimiters (shown here in blue), and computes an *expansion*, i.e. another external expression. In this case, `pattern` generates the following expansion (written concretely):

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

The definition of `pattern` looks like this:

```
syntax pattern(Q : PATTERN) at Q.t {
  static fn (body :: Body) :: Exp => (* pattern parser here *)
}
```

This TSM definition first identifies the TSM as `pattern`, then specifies a module parameter, `Q`, which must match the signature `PATTERN` (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. `P` above). The type annotation **`at`** `Q.t` specifies that all expansions that arise from the application of this TSM to a module and a delimited form must necessarily be of type `Q.t`, where `Q` refers to that module.

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. This parse function is a static function of sort `Body -> Exp`. Both of these sorts are defined in the Verse *prelude*, which is a set of definitions available ambiently. The sort `Body` gives the static function access to the body of the delimited form (e.g. the characters in blue above) and the sort `Exp`, mentioned also in the discussion of macros above, encodes the abstract syntax of external expressions (there are also encodings of other syntactic classes, e.g. types, that can appear in expressions). The parse function must treat the TSM parameters parametrically, i.e. it does not have access to any values in the supplied module parameter. Only the expansion the parse function generates can refer to module parameters.

To support splicing syntax as described in Sec. 4.2.2, the parse function must be able to extract external subexpressions directly from the supplied body. For example, consider the client code below:

```
(* TSMs can be partially applied and abbreviated *)
let syntax pat = pattern P
let val ssn = pat /\d\d\d-\d\d-\d\d\d\d/
fun example_tsm(name: string) =>
  pat /@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, it can mark it as such and insert it directly into the expansion. For example, the expansion generated for the body of `example_tsm` above would, if written concretely with spliced expressions marked, be:

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

The hygiene mechanism then ensures that only these marked portions of the generated expansion can refer to the variables at the use site, preventing inadvertent variable capture by the expansion.

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

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

Formalization To introduce the formal specification of TSMs, let us consider the rule in the EL's semantics for handling TSM application to a delimited form. To simplify matters, we will first consider only unparameterized TSMs (i.e. TSMs defined at a single type, e.g. the datatype `Pattern`, rather than a parameterized family of types):

(syn-aptsm)

$$\frac{\vdash \psi \mathbin{@} \sigma_{\text{ty}}\ \{\sigma_{\text{parser}}\} \qquad b \downarrow \sigma_{\text{body}} \qquad \sigma_{\text{parser}}(\sigma_{\text{body}}) \Downarrow \sigma_{\text{exp}} \qquad \sigma_{\text{exp}} \uparrow e_{\text{exp}} \qquad \Gamma; \emptyset \vdash e_{\text{exp}} \Leftarrow \sigma_{\text{ty}} \leadsto \iota_{\text{trans}}}{\Gamma_{\text{out}}; \Gamma \vdash \mathbf{aptsm}(\psi; b) \Rightarrow \sigma_{\text{ty}} \leadsto \iota_{\text{trans}}}$$

Looking at the conclusion of the rule, we see that TSM application takes the abstract form $\mathbf{aptsm}(\psi; b)$. The metavariable ψ ranges over *TSM expressions* and b ranges over bodies of delimited forms (i.e. the sequences of characters between the delimiters in the examples above). TSM expressions consist only of TSM definitions here (cf. the definition of `pattern` above), though when we formalize parameterized TSMs, we will distinguish TSM definitions from TSM expressions, which involve parameter application.

Note also that we omit various contexts relevant only to other Verse primitives. We include only *typing contexts*, Γ, which map variables to types in the standard way. We describe why there are two contexts – the *outer context* and the *current context* – below.

The premises can be understood as follows, in order:

1. The first premise can be pronounced "TSM expression ψ defines a valid TSM at type σ_{ty} with parse function σ_{parser}". The "outputs" of this judgement are the type σ_{ty} (of sort Ty) and the static parse function, σ_{parser} (of sort $\mathsf{ParseStream} \rightarrow \mathsf{Exp}$), defined by the TSM.

2. The second premise creates an encoding of the body of the delimited form, σ_{body}, of sort Body.

3. The third premise applies the parse function to the parse stream to generate a static representation of the expansion, σ_{exp} (of sort Exp).

4. The fourth premise decodes σ_{exp}, producing the expansion itself, e_{exp}.

5. The final premise *validates* the expansion by analyzing it against the type σ_{ty} specified by the TSM. The current typing context is "saved" for use when a spliced subexpression is encountered during this process by setting it as the new *outer context*, as indicated.

   Variables in the outer context are not directly available to the expansion, e.g. they are ignored by the (syn-var) rule:

   (syn-var)

   $$\frac{}{\Gamma_{\text{out}}; \Gamma, x : \sigma \vdash x \Rightarrow \sigma \leadsto x}$$

Outer contexts are switched back in when encountering marked spliced expressions:

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

This premise of (syn-aptsm) thus enforces both type safety and hygiene.

4.3.2 Type-Specific Languages (TSLs)

With TSMs, we achieved the conventional syntax for regular expression patterns within delimiters, but we still explicitly had to apply the TSM and provide the module parameter at each application site. To further lower the syntactic cost of using TSMs, so that it compares to the syntactic cost of derived syntax built in primitively, Verse also allows library providers to associate a TSM directly with an abstract or declared type (or type constructor, more generally). When the semantics encounters a delimited form not prefixed by a TSM name, the TSM associated with the type it is being analyzed against (i.e. that type's *type-specific language*) is implicitly applied

For example, a module P can associate `pattern` with `P.t` by qualifying the definition of the opaquely ascribed module it is defined by as follows:

```
module P = mod {
  type t = (* ... *)
  (* ... *)
} :> PATTERN with syntax pattern
```

This definition associates the TSM `pattern P` with the abstract type `P.t`. As such, the following function is equivalent to `example_tsm` (note the return type annotation, which is now necessary):

```
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 datatype, we can define the TSL directly upon declaring the list type:

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

Together with the TSL for patterns, this allows us to write a list of patterns like this:

```
let val x : list(P.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.

Abstract Types Let us examine how this mechanism works for abstract types in more detail. When sealing a module expression with a signature, the programmer can specify a previously defined TSMs for each abstract type that arises from the ascription. Each must be parameterized by a module satisfying the specified signature.

For parameterized modules, the situation is similar. For example, consider a trivially parameterized module that creates modules sealed against `PATTERN`:

```
module F() => mod {
  type t = (* ... *)
  (* ... *)
} :> PATTERN with syntax pattern
```

Each application of `F` generates a distinct abstract type. The semantics associates the appropriately parameterized TSM with each of these:

```
module F1 = F() (* F1.t has TSL pattern(F1) *)
module F2 = F() (* F2.t has TSL pattern(F2) *)
```

For a parameterized module that takes non-trivial parameters, we need only define the TSM to also take the same parameters and apply them explicitly within the definition of the parameterized module. For example, let us define 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)
```

Note that within the definition of `G`, type `u` does not have a TSL available to it.

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 *)
```

4.3.3 Limitations

Though expansion validation ensures that an incorrect parser cannot threaten type safety, proving type safety inductively is subtle because the expansion is not a subexpression of the expression it will replace. There is a "simple" solution to this problem: we impose the restriction that the expansion cannot itself use TSMs or TSLs. This would be somewhat awkward in practice, so we plan to briefly discuss other more flexible solutions as well.

To provide the strongest metatheoretic guarantees, we must have that the SL is a total functional language. Allowing non-termination in the SL would cause typechecking of the EL to become undecidable because macro expansion occurs during typechecking (typechecking the IL would remain decidable, however). Related concerns would arise if the static language supported external (e.g. I/O) effects – one would generally not want the mere act of parsing and typechecking a program to have an arbitrary effect on, for example, the programmer's file system. Mutation effects could, however, be introduced, because the static language is distinct from the external language. No static values can be "promoted" into the external language.

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.

We have discussed only expression syntax here, but have not considered pattern matching. In fact, we conjecture that it is completely straightforward to extend this work to allow a TSM to define both an expression parser and a pattern parser. Similarly, one could imagine defining TSMs at the level of modules, and at the level of types. We plan to discuss these in the dissertation. Formally specifying these will be left to future work, as it would introduce a rather large amount of technical overhead for minimal conceptual gain.

4.4 Timeline and Milestones

We described and gave a more detailed formal specification of TSMs in a recently published paper [39], and TSLs in a paper published last year [38], both in the context of the Wyvern 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 closely following this presentation, updating it only to ensure that it is consistent with the description above and with the work in the next section. The rule (syn-aptsm) above represents the first and most substantial step of this plan. This is complete on paper, so I only need to typeset it.

Wyvern does not specify an ML-style module system (type declarations are the only form of type generation) and at the time, it did not support parameterized types, so to actually support the examples we have given in this section, we then need to update the formalization to handle parameterized TSMs. I plan to flesh this out in the course of writing the dissertation, and I anticipate it requiring a further 1 month of my time after the proposal is complete. I propose the following concrete milestones:

1. equip the EL with an appropriately structured module context **(complete on paper, needs to be typeset)**
2. formally specify the syntax of parameterized TSM expressions **(complete on paper, needs to be typeset)**
3. formally specify an updated version of the TSM expression validation judgement that appeared as the first premise of (syn-aptsm) above **(complete on paper, needs to be typeset)**
4. update the metatheory established in the work cited above to handle this

To validate our claims of expressiveness, I plan to give several more examples drawn from existing languages. I will include those we have worked out in the papers just cited as well as additional examples that highlight the benefits of integration with Verse's ML-based module system. For example, I will discuss implementing Haskell's primitive "do" syntax using TSLs together with a modular encoding of monads (as described on Prof. Harper's blog [23]). I will also show how parser generators and quasiquotation (e.g. as in Scala [45] or Lisp [3]) can be seen as modes of use of TSMs/TSLs. I will also briefly summarize the corpus analysis we conducted in [38] where we found a large number of situations in large publicly available Java libraries where dynamic string parsing was apparently being used to control syntactic cost. I anticipate that writing these examples will take an additional 1 week of work, bringing the total for this portion of the dissertation up to about 5 weeks of writing (plus time for review by the committee).

5 Modularly Programmable Type Structure

In the previous section, we assumed that the EL had an otherwise standard, ML-like type structure and considered only TSMs and TSLs as departures from ML. In fact, constructs like datatypes (shown briefly in use in the previous section), tuples, records and objects are not built primitively into Verse. Instead, they can be expressed using *metamodules* (which are distinct from, though closely related to, modules, as we will discuss below). Only functions are built primitively into the Verse EL.

This section is organized like Sec. 4. We will begin with examples of type structure that we might want to express in Sec. 5.1, then discuss how existing approaches are insufficient in Sec. 5.2. Next, we outline how metamodules (together with TSMs and TSLs) serve to address these problems in Sec. 5.3 and conclude with a timeline in Sec. 5.4.

5.1 Motivating Example: Labeled Tuples and Regular Strings

As a simple introductory example showing the sorts of types and operators we might like to have available to us as clients of a language, let us define a simple *labeled tuple type* for conference papers:

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

The `let type` construction defines a synonym, `Paper`, for a type constructed by applying the *type constructor* (or *tycon*) `ltuple` to a *component specification*, which is a statically valued (cf. Sec. 2) ordered finite mapping from labels to types, written using conventional concrete syntax.

The first component is labeled `title` and specifies the *regular string type* `rstring /.+/`. Regular string types are constructed by applying the tycon `rstring` to a statically valued regular expression pattern, again written here using conventional concrete syntax like that discussed in the previous section. The second component of type `Paper` is labeled `conf` and specifies a regular string type with two parenthesized groups, corresponding to a conference abbreviation and year.

Clients can introduce regular strings in analytic position using standard string literal syntax, as long as the body of the literal is in the corresponding regular language. Clients can introduce a value of type `Paper` in an analytic position in one of two ways. We can omit the labels and provide component values positionally:

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

Alternatively, we can include the component labels explicitly for readability or if we want to give the components in an alternative order:

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

Given a value of type `Paper`, we can project out a component value by providing a label:

```
let val exmpl_conf = # <conf> exmpl
```

Here, `#` identifies an *operator constructor* (or *opcon*) parameterized by a statically valued label, written literally here as <conf>. The resulting *operator*, `# <conf>`, is then passed one *argument*, `exmpl`. An important point is that this operator is not a function, i.e. it cannot be given function type, because it is able to operate on values of *any* labeled tuple type that has a component labeled `conf`, not just `Paper`. The argument type's component specification determines the type of the operation as a whole, here `rstring /([A-Z]+) (\d\d\d\d)/`.

We can project out the first captured group from the regular string `exmpl_conf` using the `#group` operator constructor, which is parameterized by a statically valued natural number referring to the group index:

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

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

We will equip labeled tuple types with an extended set of operators that go beyond the introduction and projection operators just discussed. For example, two labeled tuples can be concatenated (with common components updated with the value on the right) using the `ltuple+` operator. Using this operator, we can add a component labeled `authors` to `exmpl`:

```
let val a : ltuple {authors : list(rstring /.+/)} = {["Harry Q. Bovik"]}
let val exmpl_paper_final = ltuple+ exmpl a
```

We can drop a component using the operator constructor `ltuple-`, which is parameterized by a static label:

```
let val exmpl_paper_anon = ltuple- <authors> exmpl_paper_final
```

There are many other operators that we might wish to describe, for both labeled tuples [7] and regular strings [18], but for our purposes it suffices to stop here.

5.2 Existing Approaches

Before describing how we express the type structure of labeled tuples and regular strings in Verse using metamodules, let us consider some alternative approaches for each.

5.2.1 Labeled Tuples

Recall that Verse builds only nullary and binary products into the IL, so we cannot express the type structure described above for labeled tuples directly using IL primitives. The Verse EL does not build in even these, but so as to separate concerns, let us assume for the moment that regular string types are available in the EL:

```
(* type synonyms for concision *)
let type title_t = rstring /.+/
let type conf_t = rstring /([A-Z]+) (\d\d\d\d)/
```

Modules It is reasonable to ask if it is possible to express the type structure of any particular labeled tuple type using the module system. For our example type `Paper` from above, we might start by defining the following signature:

```
signature PAPER = sig {
  type t
  (* introduction without labels *)
  val intro : title_t -> conf_t -> t
  (* introduction with explicit labels, in primary order *)
  val intro_title_conf : title_t -> conf_t -> t
  (* projection *)
  val prj_title : t -> title_t
  val prj_conf : t -> conf_t
}
```

Here, we've taken all possible valid introductory and projection operators and called for their expression as functions. Expressing labels explicitly is an important facet of having labeled tuple types in a language, so we put them into the function identifiers. This is, however, suboptimal because in client code, the argument values will not appear adjacent to the corresponding labels.

Even with our minimal EL, we could implement the semantics of the corresponding operators against this signature using a Church-style encoding (the details are standard and omitted here). Alternatively, if we slightly relax our insistence on minimalism and add binary products to the EL, we could use them to implement our desired semantics like this:

```
module Paper : PAPER = mod {
  type t = title_t * conf_t
  fun intro title conf => (title, conf)
  fun intro_title_conf title conf => (title, conf)
  fun prj_title (title, conf) => title
  fun prj_conf (title, conf) => conf
}
```

There are several fundamental problems with this approach. First, not every module matching the signature `PAPER` correctly implements the semantics of the operators we seek to express, so each such module would itself need to be verified. In particular, we need to verify the universal condition that for all `e : Paper.t` and `f : Paper.t -> T`, we have that `f(e)` is equivalent to `f(Paper.intro (Paper.prj_title e) (Paper.prj_conf e))`. The volume of boilerplate code that needs to be generated and verified manually is factorial in the number of components of the labeled tuple type we are expressing (because we must encode each permutation of label orderings with a distinct introductory function).

Even if we were to automate this (using, for example, *type macros* as in Scala [6], adapted to support Verse's modules), another issue is that we can only reasonably express the introductory and projection operators by specifying their semantics as functions like this. To enumerate all possible operators that arise from the opcon `ltuple-` using functions would require constructing an encoding of every labeled tuple type that might arise from dropping one or more components from the labeled tuple in question. If fully enumerated, this would add an exponential factor to our volume of boilerplate code. Finally, there is simply no way to specify the semantics `ltuple+` with a finite collection of functions because there are infinitely many extensions of any labeled tuple type.

Records and Tuples If, instead of this module-based encoding, we further weaken our insistence on minimalism and primitively build record and tuple types into the EL, as SML and many other languages have done, let us consider what might be possible. Note that these too are library constructs in Verse, but because they are so commonly built in to other languages, this situation is worth considering.

The simplest approach we might consider taking is to directly map each labeled tuple type to a record type having an identically written component specification:

```
let type Paper_rcd = record {
  title : title_t
   conf : conf_t
}
```

Unlike labeled tuple types, record types are identified up to component reordering, so this mapping does not preserve certain type disequalities between labeled tuple types. Without positional information available in the type, it is not possible to allow clients to omit component labels when introducing a value of record type.

To work around this limitation if this is the only reason we care about preserving these type disequalities, we could define two conversion functions involving unlabeled tuples:

```
fun Paper_of_tpl (t : title_t, c : conf_t) =>
  {title => t, conf => c}
fun tpl_of_Paper {title => t, conf => c} =>
  (t, c)
```

For clients to be able to rely on this approach, we must extrinsically verify that the library provider implemented these functions correctly. Clients must also bear the syntactic and dynamic costs of explicitly invoking these functions. We could define a TSM for every such type to reduce this cost to clients, albeit at an additional cost to providers.

If, on the other hand, we do care about preserving all type disequalities between labeled tuple types, the workaround is even less elegant. We first need to define a record type with component labels tagged in some sufficiently unambiguous way by position, e.g.:

```
let type Paper = record {
  __0_title : title_t
  __1_conf : conf_t
}
```

To avoid exposing these labels directly to clients, we need two auxiliary type definitions:

```
let type Paper_tpl = (title_t * conf_t)
let type Paper_rcd = (* as above *)
```

and four conversion functions:

```
fun Paper_of_tpl (t : title_t, c : conf_t) => {
  __0_title => t,
  __1_conf => c
}
fun tpl_of_Paper {__0_title => t, __1_conf => c} =>
  (t, c)
fun Paper_of_rcd {title => t, conf => c} => {
  __0_title => t,
  __1_conf => c
}
fun rcd_of_Paper {__0_title => t, __1_conf => c} = {
  title => t,
  conf => c
}
```

Clients once again bear the costs of applying these conversion functions ahead of every operation and providers again bear the burden of defining this boilerplate code (which has volume linear in the number of components, albeit with a constant factor that is again not easy to dismiss) and verifying that these functions were implemented correctly.

As with the modular encoding of labeled tuples we attempted above, neither of these encodings using records and unlabeled tuples provides us with any way to uniformly express the more interesting operations, like `ltuple+` or `ltuple-`, that we discussed earlier.

5.2.2 Regular Strings

Let us now turn to regular string types. Note that we have specified the full semantics of regular string types in a recent workshop paper [18].

Dynamic Checks The most common alternative to regular strings in existing languages is to simply use standard strings (which themselves may be expressed as lists or vectors of characters together with derived syntax) and insert dynamic checks around operations to maintain the regular string invariant. This clearly does not express the static semantics of regular strings, and incurs syntactic and dynamic cost.

Type Refinements We might attempt to recover the static semantics of regular strings by moving the logic governing regular string types into an external system of *type refinements* [17]. Such systems allow programmers to specify *refinements* of a type that capture more specific invariants. For example, we might consider capturing the regular string invariant using refinements of the `string` type. Refinement annotations can then be processed by external refinement checkers, like SML CIDRE [10]. In Verse, we might supply refinement specifications using structured comments:

```
  let val conf : string (* ~ rstring /([A-Z]+) (\d\d\d\d)/ *) = "EXMPL 2015"
```

One immediate limitation of this approach is that there is now no way to express the group projection operator described above, i.e. we cannot define a function `#group` that can be applied like this:

```
  let val conf_venue = #group 0 conf
```

The reason is that group projection does not have a function-like semantics – it needs access to information that here is available only in the type refinement of each regular string, i.e. the definitions of the captured groups. To access a group, we would instead need to dynamically match the string against the regular expression explicitly:

```
  let val conf_venue = nth 0 (match /([A-Z]+) (\d\d\d\d)/ conf)
```

This has higher syntactic and dynamic cost. Moreover, proving that there will not be an out-of-bounds error in expressions like this requires more sophisticated reasoning about dynamic behavior.

Another consideration is that type refinements do not introduce type disequalities. As a result, a compiler cannot use the invariants they capture to optimize the representation of a value. For example, the fact that some value of type `string` can be given the refinement `rstring /A|B/` cannot be justification to locally alter its representation to, for example, a single bit, both because it could later be passed into a function expecting a standard string and because there are many other possible refinements. Another perhaps more useful optimization that is not possible is to represent regular strings that have captured groups in a manner that ensures that group projection has constant dynamic cost (by running the match once when the string is introduced and caching the group values). Another example where this is relevant is when a language designer considers omitting a primitive type of "non-negative integers representable in 32 bits" because this seems "merely" to be a refinement of (mathematical) integers. But by eliminating the structural distinction between the two types, compilers lose the ability to use a machine representation more suitable to this particular static invariant. Of course, such structural distinctions are sometimes inconvenient because they require defining and applying coercions with possibly non-zero dynamic cost to go from values of one type to another. Nevertheless, it is certainly not the case that a language can reasonably predict all possible structural distinctions that might be useful *a priori*.

5.3 Contributions

Verse introduces a *metamodule system* that gives library providers more direct control over the type structure of the EL. Using the metamodule system, library providers can express all of the types and operators discussed above.

Just as the Verse (i.e. ML) module system is organized around *signatures* and *modules*, the Verse metamodule system is organized around *metasignatures* and *metamodules*. These are most easily introduced by example. We consider only labeled tuple types in this section (regular string types will follow analagously; we will discuss regular string types in more detail in the dissertation).

5.3.1 Example: Labeled Tuple Types via Metamodules

Figure 2 defines a metasignature, `LTUPLE`, and a matching metamodule, `Ltuple`. It also defines a *metamodule-parameterized TSM* `ltpl`, which will give us the introductory syntax shown in Sec. 5.1. The remainder of this subsection explains this figure.

Type Constructors On line 2 of Figure 2, the metasignature `LTUPLE` specifies that all matching metamodules must define a type constructor `c` parameterized by static values of sort `ComponentSpec`. Let us assume that we can write static values of sort `ComponentSpec` concretely like this (in the dissertation, we will discuss how it would be possible to adapt TSLs to also operate at the level of static values):

```
let static val Paper_tyidx :: ComponentSpec = {
  title : rstring /.+/,
   conf : rstring /([A-Z]+) (\d\d\d\d)/
}
```

Applying the type constructor `Ltuple.c` to this parameter gives us a type:

```
let type Paper = Ltuple.c Paper_tyidx
```

Line 60 defines the tycon synonym `ltuple` for `Ltuple.c`. Substituting into the above, we recover the definition of `Paper` given at the beginning of Sec. 5.1:

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

Note that because types are static values of sort `Ty`, this is equivalent to writing:

```
let static val Paper :: Ty = ltuple {
  title : rstring /.+/,
   conf : rstring /([A-Z]+) (\d\d\d\d)/
}
```

Also note that to ensure that type equality is decidable, tycon parameters can only be of a sort for which equality coincides with structural comparison of values. We call these sorts *equality sorts*. Equality sorts are exactly analagous to equality types as found in Standard ML. The main implication of this restriction is that type parameters cannot contain static functions (because they cannot be compared structurally).

Type Translations Recall from Sec. 3 that each external type must translate to an internal type. Each metamodule controls the translation of types constructed by its tycons by defining a static function called the *type translator*. This static function programmatically generates the type's translation given its type parameter. Type translations are represented as static values of sort `ITy`, which we assume supports standard *quasiquotation* syntax (e.g. as in Scala [45] or Lisp [3]).

The type translator for `Ltuple.c` is defined on lines 20-26 of Figure 2. We have chosen here to translate labeled tuple types to nested binary product types internally. The type translator generates these by folding over the component mapping (using helper function `comp_spec_to_tuples :: ComponentSpec -> list(Lbl * Ty)`, not shown). It generates the representation of `unit` for the empty case, and nests the binary product types in the recursive case (using the standard *unquote* syntax, `%`). For example, the representation of the type translation of `Paper` determined by this static function is:

```
‘trans(rstring /.+/) * (trans(rstring /([A-Z]+) (\d\d\d\d)/) * unit)‘
```

Notice that the translations of the component regular string types are not inserted directly, but are rather treated parametrically using the *translational form* `trans(T)`. This will be key to the modularity principle – *translation independence* – that we will discuss shortly.

Let us assume, for example, that all regular string types translate to standard internal strings, encoded by some internal type abbreviated `string`. After we substitute this in for these translational forms, we arrive at the translation of `Paper`:

```
string * (string * unit)
```

Formally, we should be able to derive that $\vdash \texttt{Paper}\ \texttt{type} \leadsto \texttt{string} \times (\texttt{string} \times \mathsf{unit})$. The following rule governs type translation:

$$\frac{\begin{array}{ccc}\vdash_\Phi \textbf{ty}(m \cdot c; \sigma_{\text{param}}) :: \mathsf{Ty} & \textbf{ty}(m \cdot c; \sigma_{\text{param}})\ \texttt{sval}_\Phi & \textbf{translator}(\Phi; m \cdot c) = \sigma_{\text{translator}} \\ \sigma_{\text{translator}}(\sigma_{\text{param}}) \Downarrow \sigma_{\text{trans}} & \sigma_{\text{trans}} \uparrow^m_\Phi \tau_{\text{abstrans}} \parallel \delta : \Delta & \Delta \vdash \tau_{\text{abstrans}}\ \texttt{itype}\end{array}}{\vdash_\Phi \textbf{ty}(m \cdot c; \sigma_{\text{param}})\ \texttt{type} \leadsto [\delta]\tau_{\text{abstrans}}}\ \text{(trans-ty)}$$

Note that this rule is again slightly simplified here. It would need to be modified to handle external type abstraction, for example. However, it is pedagogically useful to begin with this simpler rule. Note that our purpose in this proposal is not to give a complete formal account of our contributions, but only to sketch out what this formal account will look like.

Looking at the conclusion of the rule, we see that in the abstract syntax of the SL, types constructed by user-defined tycons take the form $\textbf{ty}(m \cdot c; \sigma_{\text{param}})$, where $m \cdot c$ refers to a tycon c defined by metamodule m and σ_{param} is the type parameter. The premises can be understood as follows:

1. The first premise checks that the type is of the required sort, Ty. This involves finding the parameter sort associated with the tycon in the *metamodule context*, Φ (we omit the straightforward definitions and rules for concision here).

2. The second premise checks that the parameter is a static value.

3. The third premise looks up the translator associated with $m \cdot c$ in Φ.

4. The fourth premise applies this translator to the type parameter to generate the representation of the type translation, σ_{trans} (e.g. shown above for `Paper`).

5. The fifth premise decodes σ_{trans} to produce an *abstracted type translation*, τ_{abstrans}. It is "abstracted" in that translational forms referring to types constructed by metamodules other than m, e.g. `trans(rstring /.+/)` above, have been replaced by type variables. For example, the representation above produces the abstracted typed translation $\alpha_1 \times (\alpha_2 \times \mathsf{unit})$. The actual translations of these types are packaged into a standard *type substitution*, δ. For example, here $\delta = [\texttt{string}/\alpha_1, \texttt{string}/\alpha_2]$. The context corresponding to this substitution, here $\Delta = \alpha_1, \alpha_2$, is also generated.

6. The final premise validates the abstracted type translation under Δ by making sure it is a valid internal type according to the internal statics.

7. Finally, the conclusion of the rule applies δ to the abstracted translation to produce the final type translation.

We will see why having this ability to selectively hold type translations abstract is critical to modularity after we consider operator constructors.

Operator Constructors On line 3 of Figure 2, the metasignature LTUPLE specifies an operator constructor, intro_unlabeled, parameterized by static values of sort unit. The qualifier **ana** specifies that this opcon is only for use in analytic positions (i.e. where the expected type is known). For example, we can apply Ltuple.intro_unlabeled as follows, because the return type of the function determines the expected type:

```
fun make_paper(c : conf_t, t : title_t) : Paper =>
  Ltuple.intro_unlabeled () (c; t)
```

First, we provided the opcon with a static parameter value of the appropriate sort, here unit, to create an operator. Then, we supply the operator with an *argument list*, (conf; title).

This is syntactically unwieldy. Luckily, we can define a parameterized TSM, ltpl, that gives us a more conventional syntax. As shown on lines 10-17 of Figure 2, ltpl defines syntax at all types that are of the form L.c(param), where L is a metamodule matching LTUPLE and param is a static value of sort ComponentSpec. In other words, the TSM defines syntax for all labeled tuple types. We elide the parser implementation, but as the comment indicates, we can rewrite make_paper like this:

```
fun make_paper_tsm(c : conf_t, t : title_t) : Paper =>
  ltpl Ltuple Paper_tyidx {c, t}
```

To avoid having to explicitly name the TSM and supply the parameters, we designate ltpl as Ltuple.c's TSL on line 58. Now we can rewrite make_paper in the standard way:

```
fun make_paper_tsl(c : conf_t, t : title_t) : Paper =>
  {c, t}
```

Putting syntax aside, the metamodule defining the opcon, Ltuple, programmatically determines how to typecheck and translate this operation, again by the defining a static function called the *opcon definition*. The definition of intro_unlabeled is shown on lines 28-37 of Figure 2. For analytic opcons like intro_unlabeled, the semantics provides the opcon definition with the type that the expression is being analyzed against, here Paper, the operator parameter, here (), and a list of *argument interfaces*, which will allow it to ask the semantics to recursively typecheck and translate the arguments provided in the argument list. It must return a representation of an internal expression, which will become the translation of the operation. Internal expressions are represented as static values of sort IExp, and again we assume that we can use standard quasiquote syntax. The remaining opcons (which we will not consider in detail here) are synthetic opcons, which means they can be used anywhere. Rather than taking a type as input, they produce a pair of a type and a translation as output.

On lines 28-37, the opcon definition first checks that the type is actually a labeled tuple type using the **tycase** primitive, which case analyzes against the type constructor of the provided type. There must always be a default case to ensure exhaustiveness. In this example, the default case raises a type error (with an error message, elided here).[10] If the tycon is c, we pair up the items in the type's component specification with the arguments and fold over this list. The empty case generates the representation of the trivial internal value and the recursive case generates nested pairs. Notice that this follows the structure of the type translator for c, reflecting the fact that introductory operations at a type must generate a translation consistent with the corresponding type translation for type safety to hold. The static operation **ana**(arg, ty) requests that the provided argument be analyzed against the provided type, evaluating to a representation of its translation if this succeeds and raising an error if it fails. For our example above, the representation of the translation generated by the opcon definition is:

```
‘(anatrans[0](conf_t), (anatrans[1](title_t), ()))‘
```

[10] The reason why we use this exception-like data flow for raising type errors, rather than using an option sort, is somewhat subtle so we defer it for discussion in the dissertation.

An *argument translation* of the form `anatrans[i](T)` stands in for the translation of argument `i` analyzed against type `T`.

The semantics next generates an *abstracted translation*, which is a corresponding internal term where each argument translation has been replaced by a corresponding fresh variable. To validate it, we assume that the type of each variable is the abstracted type translation of the type the corresponding argument was analyzed against. So in this case, the abstracted translation is $(x_0, (x_1, ()))$ and it will be validated against the abstracted type translation for `Paper`, i.e. $\alpha_0 \times (\alpha_1 \times \texttt{unit})$, under a context where $x_0 : \alpha_0$ and $x_1 : \alpha_1$ (this will, of course, succeed). Only after this succeeds are the actual argument translations substituted in to generate the final operation translation, here $(c, (t, ()))$.

By treating the arguments to the operation parametrically in this way, we can be sure that the metamodule defining labeled tuple types cannot violate the translation invariants that other metamodules maintain. For example, even if regular strings are implemented (by some other metamodule, not shown) as strings satisfying the regular string invariant, `Ltuple` knows only that *there exists* some translation for each regular string type, but nothing more. Thus, it cannot define “bad” opcons like this:

```
syn opcon bad {
  static fn (op_param :: unit, args :: list(Arg)) :: Ty * IExp =>
    (rstring /.+/, ‘""‘)
}
```

Here, the opcon ignores the arguments and synthesizes a regular string type that the client will assume classifies non-empty strings. However the corresponding translation is in fact the empty string. So even if all of the opcons defined in the metamodule implementing regular strings maintained this regular string invariant, this opcon would violate it. Note that simply checking the translation is well-typed would not solve this problem. The operator translation (the empty string) is consistent with the type translation (`string`). Fortunately, by holding the type translation of regular strings abstract when validating the output of this opcon, the problem is caught – the empty string is not of type α. Put another way, the problem with this opcon is that it is not defined in a translation-independent manner with respect to tycons defined in another metamodule. This is exactly analagous to the representation independence property underlying modular reasoning with ML-style modules. By leveraging type abstraction, we are able to achieve modularity guarantees without requiring mechanized proofs of translation invariants (just as ML does not require mechanized proofs of representation invariants).

To formalize the intuition just described, we will need to introduce a non-trivial number of technical devices. It would thus require too much space to give a comprehensive formal account of this mechanism in this proposal, so we leave this as work that remains to be done (though see below for progress). In broad strokes, it is similar to the rule for type translation described above in that we validate the operation translation before applying the substitutions for type and expression variables.

Note that the static functions that define these opcons contain code that looks just like the code that would appear in a standard implementation of the first stage of a compiler for a language that built in labeled tuple types. The novelty is not in these definitions, but in how they are organized and brought into the language.

5.3.2 Limitations

The main limitation of the mechanism just described is that it only supports defining types and operators that do not require adding new contexts to the EL typing judgement. No new binding structure can be defined. Fortunately, a number of interesting examples found in existing languages have this character, including labeled tuple types, record types, labeled sum types, object types (of various designs), regular string types and range-restricted numeric types. We plan to detail these in the thesis. On the other hand, constructs like ML-style exception values, OCaml-style open datatypes or Java-style class-based object systems would not be possible because they require statically tracking which constructors/classes are in scope. We plan to discuss adding support for new user-defined static contexts as an avenue for future work in the dissertation.

Establishing the metatheory of metamodules is also somewhat tricky, again because it involves reasoning about expressions that are not clearly subexpressions of those in the conclusion of the relevant rules. We plan to give an induction principle by which we can prove the necessary metatheorems in the dissertation.

5.4 Timeline and Milestones

The work above has recently been refactored so as to more closely follow the construction of the ML module system. Previously, the main organizational unit was a single tycon, rather than a metamodule. In this alternative form, we have a complete paper draft and a nearly complete technical report that gives the details of all of the mechanisms described above. They can be found at:

- Paper:
  `https://github.com/cyrus-/papers/blob/master/atlam-icfp15/atlam-icfp15.pdf`
- Technical Report:
  `https://github.com/cyrus-/papers/blob/master/atlam-icfp15/atlam-icfp15-supplement.pdf`

We propose the following concrete milestones, to be completed after the work in the previous section:

1. Refactor the syntax around metamodules **(1 week)**
2. Update the static language's semantics **(1 week)**
3. Update the external language's semantics **(1 week)**
4. Update and complete the metatheory **(3 weeks)**
5. Add support for external type abstraction **(1 week)**

To further validate our work, we plan to detail a number of the examples mentioned above, including regular strings, datatypes, and a simple prototypic object system. We anticipate that the writing involved to do this, along with other writing tasks, will take another **2 weeks**. So the total for this section is 9 weeks.

6 Conclusion

In summary, we have proposed a new functional programming language, Verse, that gives programmers the ability to programmatically implement new syntactic expansions and new type structure using statically evaluated functions. Consequently, Verse does not need to build in constructs that comparable languages need to (or would need to) build in, like regular expression syntax, list syntax, labeled tuples and regular strings. These and other examples that we will cover in the dissertation will serve to validate our claim that Verse is highly expressive (and thus should lead to a decreased need for dialect formation).

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 course of 17 weeks, per the timelines in the sections above. Given that the proposal should be complete by the end of Summer 2015, this implies that the remaining work can be completed over the course of Fall 2015. Final updates and the presentation of the dissertation should then be in early Spring 2016.

```
metasignature LTUPLE = metasig {
  tycon c of ComponentSpec
  ana opcon intro_unlabeled of unit
  syn opcon intro_labeled of list(Lbl)
  syn opcon # of Lbl
  syn opcon + of unit
  syn opcon - of Lbl
}

syntax ltpl(L :: LTUPLE, param :: ComponentSpec) at L.c(param) {
  static fn ps => (*
    - elaborates {e_1, ..., e_n} to
        L.intro_unlabeled () (e_1; ...; e_n)
    - elaborates {lbl1 => e_1, ..., lbln => e_n} to
        L.intro_labeled [lbl1, ..., lbln] e_1 ... e_n
  *)
}

metamodule Ltuple :: LTUPLE = metamod {
  (* translate labeled tuple types to nested products *)
  tycon c {
    (* type translation generator: *)
    static fn (param :: ComponentSpec) :: ITy => fold (comp_spec_to_tuples param)
      `unit`
      (fn ((lbl, ty), cur_ty_trans) => `trans(ty) * %cur_ty_trans`)
  }

  ana opcon intro_unlabeled {
    (* translate intro operator to nested pairs *)
    static fn (ty :: Ty, op_param :: unit, args :: list(Arg)) :: IExp => tycase(ty) {
        c(ty_param) => (fold (zipExact (comp_spec_to_tuples ty_param) args)
          `()`
          (fn (((lbl, ty), arg), cur_trans) => `(%{ana(arg, ty)}, %cur_trans)`)
        )
      | _ => raise TyErr("...")
    }
  }

  syn opcon intro_labeled {
    static fn (op_param :: list(Lbl), args :: list(Arg)) :: Ty * IExp =>
      (* ... similar to intro_unlabeled but reorder first ... *)
  }

  syn opcon # {
    static fn (op_param :: Lbl, args :: list(Arg)) :: Ty * IExp =>
      (* ... generate the appropriate n-fold projection ... *)
  }

  syn opcon + {
    static fn (op_param :: unit, args :: list(Arg)) :: Ty * IExp =>
      (* generate the new type and combine the two nested tuples *)
  }

  syn opcon - {
    static fn (op_param :: Lbl, args :: list(Arg)) :: Ty * IExp =>
      (* generate the new type and drop the appropriate component *)
  }
} with syntax ltpl
(* synonyms used in Sec. 5.1 *)
let tycon ltuple = Ltuple.c
let opcon # = Ltuple.#
let opcon ltuple+ = Ltuple.+
let opcon ltuple- = Ltuple.-
```

Figure 2: Using metamodules to define the type structure of labeled tuple types.

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] L. Cardelli and J. C. Mitchell. Operations on records. In *Mathematical Structures in Computer Science*, pages 3–48, 1991.

[8] 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.

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

[10] R. Davies. A refinement-type checker for Standard ML. In *Sixth International Conference on Algebraic Methodology and Software Technology (AMAST'97)*, pages 565–566, 1997.

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

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

[13] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, and J. van der Woning. The state of the art in language workbenches. In *Software Language Engineering (SLE '13)*. 2013.

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

[15] M. Fluet, M. Rainey, J. H. Reppy, A. Shaw, and Y. Xiao. Manticore: a heterogeneous parallel language. In *Workshop on Declarative Aspects of Multicore Programming (DAMP '07)*, pages 37–44. ACM, 2007.

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

[17] 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.

[18] N. Fulton, C. Omar, and J. Aldrich. Statically typed string sanitation inside a Python. In *First International Workshop on Privacy and Security in Programming (PSP '14)*, 2014.

[19] M. Furr and J. S. Foster. Checking type safety of foreign function calls. In *PLDI '05*, pages 62–72, 2005.

[20] 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.

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

[22] 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.

[23] R. Harper. Of Course ML Has Monads! `https://existentialtype.wordpress.com/2011/05/01/of-course-ml-has-monads/`. Retrieved June 06, 2015.

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

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

[26] R. Harper and C. Stone. A Type-Theoretic Interpretation of Standard ML. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

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

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

[29] 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).

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

[31] 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.

[32] 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.

[33] D. MacQueen. Modules for Standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 198–207, 1984.

[34] L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *PPDP '05*, pages 82–93. ACM, 2005.

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

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

[37] 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.

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

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

[40] 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.

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

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

[43] 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.

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

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

[46] 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.

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

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

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

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

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

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