Modularly Programmable Syntax and Type Structure (Thesis Proposal)

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

Abstract

Functional programming languages like ML descend conceptually from minimal lambda calculi, but to be pragmatic, expose a concrete syntax and type structure to programmers of a more elaborate design. Language designers have many viable choices at this level, as evidenced by the diversity of dialects that continue to proliferate around these languages. But language dialects cannot be modularly combined, limiting the choices available to programmers. We propose a new functional programming language, Verse, designed to decrease the need for dialects by giving library providers the ability to safely and modularly express derived concrete syntax and type structure of a variety of designs atop a minimal type-theoretic core.

1 Motivation

Functional programming languages like Standard ML (SML), OCaml and Haskell descend from mathematically elegant typed lambda calculi, diverging mainly in that they consider various pragmatic issues alongside fundamental metatheoretic issues in the design of their syntax and semantics. For example, they build in record types, generalizing the nullary and binary product types more common in minimal calculi, because explicitly labeled components can be cognitively useful to programmers. Similarly, they build in derived concrete syntax for common library constructs, like lists.

The hope is that a limited number of "embellishments" like these will suffice to turn a minimal lambda calculus into a broadly practical programming language. Unfortunately, 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. Indeed, tools that assist with the construction of so-called "domain-specific" language dialects are only becoming increasingly popular. This calls for an investigation: why is it that programmers and researchers are 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?

Perhaps the most common reason for this proliferation of dialects may simply be that the *syntactic cost* of expressing a construct of interest using contemporary general-purpose primitives is not always ideal. To decrease syntactic cost, library providers construct *derived syntactic dialects*, i.e. dialects that are specified by context-free elaboration to the existing language. For example, Ur/Web is a derived syntactic dialect of Ur (a language that itself descends from ML) that builds in derived syntax (colloquially, "syntactic sugar") for SQL queries, HTML elements and other datatypes used for web programming [5]. Many other types of data could similarly benefit from the availability of specialized derived syntax. For example, we will consider regular expression patterns expressed using abstract data types (as a mode of use of the ML module system) in Sec. 4. We will be able to express precisely

the semantics that we seek, but existing approaches, e.g. using dynamic string parsing, for approximating the concrete syntax of patterns will leave something to be desired. Tools like Camlp4 [22] and Sugar* [6, 7] have lowered the engineering costs of constructing derived syntactic dialects in situations like these, contributing to their ongoing proliferation.

More advanced dialects introduce new type structure, going beyond what is possible with derived syntax. Record types are themselves a simple example – they cannot be expressed by context-free elaboration to a language with only nullary and binary product types. Various functional languages have explored "record-like" primitives that go still further, supporting functional update and extension operators, width and depth coercions (sometimes implicit), methods, prototypic dispatch and other "semantic embellishments" that cannot be expressed by context-free elaboration to a language with only standard record types (we will detail an example in Sec. 5). OCaml primitively builds in the type structure of "polymorphic variants", "open datatypes" and operations that use format strings like `sprintf` [22]. ReactiveML builds in primitives for functional reactive programming [24]. ML5 builds in high-level primitives for distributed programming based on a modal lambda calculus [26]. Manticore [9] and AliceML [32] build in parallel programming primitives with a more elaborate type structure than is found in their respective core calculi. MLj builds in the type structure of the Java object system (motivated by a desire to interface safely and naturally with Java libraries) [2]. 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 [12]. 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 [8] lower the cost of implementing them, again contributing to their continued proliferation.

The reason why this proliferation of language dialects should be considered alarming is that it 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 MLj demonstrates, addressing the interoperability problem requires somehow "combining" the dialects into a single language. 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 syntactic and semantic properties that can be established about the dialects in isolation. For example, consider two derived syntactic dialects specified using context-free grammars, one specifying derived syntax for finite mappings, the other specifying a similar syntax for ordered finite mappings. Though each can be shown to have an unambiguous concrete syntax in isolation, when their grammars are naïvely combined by, for example, Camlp4, ambiguities arise. Due to this paucity of modular reasoning principles, language dialects are not practical for software development "in the large".

Dialect designers must ultimately take a less direct approach to have an impact on large-scale software development – they must convince the designers in control of comparatively popular languages, like Ocaml and Scala, to include some suitable variant of the primitives they espouse 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 [27]). Second, primitives introduced earlier in a language's lifespan 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. Recalling the words of Reynolds, which are as relevant today as they were almost half a century ago [31]:

> *The recent development of programming languages suggests that the simultaneous achievement of simplicity and generality in language design is a serious unsolved problem.*
>
> — John Reynolds (1970)

This leaves the subset of the language design community interested in keeping general-purpose languages small and free of *ad hoc* primitives with two possible paths forward. One, exemplified (arguably) by SML, is to simply eschew the introduction of specialized syntax and type structure and settle on the existing primitives, which can be said to sit at a "sweet spot" in the overall language design space (accepting that in some circumstances, this trades away expressive power or leads to high syntactic cost). The other is to search for more general primitives that reduce these seemingly *ad hoc* primitives 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 [37]. Using GADTs, Ocaml was able to move some of the *ad hoc* machinery for typechecking operations that use format strings, like `sprintf`, out of the language and into a library (however, syntactic machinery remains primitively built in).

2 Proposed Contributions

Our broad aim in the work being proposed is to introduce a new functional programming language called Verse[1] that takes more radical steps down the second path just described by giving library providers the ability to directly express new derived concrete syntax as well as new type structure of a variety of designs in a safe and modularly composable manner atop a minimal type-theoretic core. The result is that 1) Verse is smaller than comparable languages like ML and Scala, and 2) dialect formation is less often necessary.

We formally introduce the semantics of Verse in Sec. 3. Our novel contributions will be two primitives that eliminate the need for many others:

1. **Typed syntax macros** (TSMs), introduced in Sec. 4, eliminate the need to primitively build in derived concrete syntax 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 (at a specified type or parameterized family of types) over the parsing and elaboration of delimited segments of concrete syntax.

2. **Metamodules**, introduced in Sec. 5, eliminate the need to primitively build in the type structure of constructs like records (and variants thereof), labeled sums and other more esoteric constructs that we will introduce later by giving library providers programmatic hooks directly into the semantics. We will see direct analogies between ML-style modules (which Verse also supports) and metamodules in Sec. 5.

Both TSMs and metamodules are novel forms of *static code generation*, i.e. the relevant rules in the static semantics call for the evaluation of *static functions* that generate static representations of expressions and types. Library providers write static functions in the Verse *static language* (SL).

The key challenge in the design of these primitives will come in ensuring that they are metatheoretically well-behaved, given that they aim to decentralize control over aspects of the concrete syntax and type structure that, in other contemporary languages, are under the exclusive control of the language designer. If we are not careful, many of the problems that arise when combining language dialects, discussed earlier, could simply shift into the

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

semantics of these primitives.[2] Our main technical contributions will be in 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 strong *hygienic type discipline* and, most uniquely, powerful *modular reasoning principles*. In other words, 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 program context (i.e. *hygenically*) and in any combination, without the possibility of conflict (i.e. *modularly*).[3] 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 implement new derived concrete syntax and type structure atop a minimal 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 rather poor taste. We expect that in practice, Verse will come with a standard library defining a carefully curated collection of standard constructs, as well as guidelines for advanced users regarding when it would be sensible to use the mechanisms we introduce (following the example of languages that support operator overloading or type classes [15], which also have the potential for such "abuse"). For most programmers, using Verse should not be substantially different from using a comparable language or one of its dialects.

Finally, Verse intentionally does not support dependent types ala Coq, Agda or Idris because they lack a phase separation between "compile-time" and "run-time." Verse is designed for use for programming tasks where SML, Ocaml, Haskell or Scala would be used today. That said, we conjecture that the primitives we describe could be added to languages like Gallina (the "external language" of the Coq proof assistant [25]) or to the "program extraction" mechanisms of proof assistants like Coq with some modifications, but do not plan to pursue this line of research here.

3 Verse

To situate ourselves within a formal framework, let us begin with a brief overview of how Verse is organized and specified. Verse consists of a *module language* atop a *core language*.

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

[3] 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.

3.1 Core Language

The Verse core language – the language of types and expressions – will be the focus of our formal efforts. 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 compare the core language to the first stage of a type-directed compiler (e.g. the TIL compiler for Standard ML [35]) shifted "one level up" into the language itself. A third sublanguage, the *typed static language* (SL), is involved in giving library providers programmatic control over aspects of this type-directed translation from the EL to the IL.

We summarize the main judgements relevant to the IL, EL and SL below. Readers who prefer to see examples first can safely skim these sections for now.

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* [18] 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**
$\vdash \iota : \tau$	Internal expression ι has internal type τ.
$\vdash \tau$ `itype`	Internal type τ is valid.

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'.
ι `val`	Internal expression ι is a value.

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

Note that features like state, exceptions, minimal concurrency primitives, scalars, arrays and other primitives 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. We wish to give a minimal account of our novel contributions, so we will stick to this simpler, well-understood IL for the purposes of this work.

3.1.2 External Language

The external language supports a richer syntax and type structure atop the IL. Throughout this work, the words "expression" and "type" used without qualification refer to external expressions and types, respectively. The main judgements in the specification of the EL take the following form (again omitting various contexts for now):

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

Notice that 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") [30]. 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 intentionally do not seek to support global type inference.

3.1.3 Static Language

Finally, the static language plays an essential role in the semantics of the novel primitives that we will introduce into the EL. The SL is another typed lambda calculus consisting of *static expressions*, σ, classified by *sorts*[4], π, according to the following judgements (the "static statics"):

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

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 a 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 external type translation judgement above, external types are static values of a primitive sort written Ty. We will return to how this works (and consider the metatheoretic implications of this choice) in Sec. 5.

3.2 Module Language

The Verse module language is taken directly from Standard ML, with which we assume a working familiarity for the purposes of this proposal [17, 23] (a related module language, e.g. Ocaml's, would work just as well for our purposes). We will give examples of its use in Sections 4 and 5, but because it has been thoroughly studied in the literature, we will defer to prior work both here and in the dissertation for most formal details. In particular, we will 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 addition of a module system. To do so, we will assume that modules are being tracked by a 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 the EL and consider its concrete syntax. Verse, like many major contemporary languages, specifies a textual concrete syntax for the EL.[5] Because the purpose of concrete syntax is to serve as the programmer-facing user interface of the language, it is common practice to build in derived syntactic forms (colloquially, *syntactic sugar*) that capture common idioms more concisely or naturally. For example, derived list syntax is built in to many functional languages, so that instead of having to write out `Cons(1, Cons(2, Cons(3, Nil)))`, the programmer can equivalently write `[1, 2, 3]`. Many languages go beyond this, building in derived syntax associated with various other types of data, like vectors (the SML/NJ dialect of SML), arrays (Ocaml),

[4] We use this word to avoid confusion with the common (though redundant) phrase "static type".

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

monadic commands (Haskell), syntax trees (Scala), 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 derived syntax of a variety of designs on their own, in a safe and modular manner. We begin in Sec. 4.1 by detailing a representative 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* and the related *type-specific languages* – and discuss how they resolve these issues. We also discuss how TSMs are specified in terms of the judgement forms 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 [36].

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

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

One way to express this abstract syntax is by defining a recursive sum type [18]. Verse supports these as case types, which are comparable to datatypes in ML (we plan to show how the type structure of case types can in fact be expressed in libraries later):

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

However, there are some reasons not to expose this representation of patterns directly to clients. First, regular expression patterns are usually identified up to their reduction to a normal form. For example, $\mathbf{seq}(\mathbf{empty}, p)$ 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 : (
    'a ->
    (string -> 'a) ->
    (t * t -> 'a) ->
    (t * t -> 'a) ->
    (t -> 'a) ->
```

```
    (t -> 'a) ->
    'a)
}
```

Client of any module `P` that satisfies `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) so that 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.

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`.[6] In a small lab study, we observed that this class of error was common, and often temporarily confused even experienced programmers if they had not used regular expressions recently [29]. The standard workaround has higher syntactic cost – one must double all backslashes:

   ```
   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 uninterpretted. 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}
   ```

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

2. 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 *)
  | _ => (* ... *)
}
```

   In other words, this approach leaves the well-formedness of patterns constructed from strings as a static verification condition. Though violations of this condition 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" [34].

   Statically verifying that this condition holds throughout a program is quite difficult to automate in general, because it requires reasoning about arbitrary dynamic behavior. In this example, a decision procedure would 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 (further discussed below), the problem is not solved by "simply" statically evaluating the expression.

   Of course, asking the client to provide a proof of well-formedness would completely defeat the purpose – providing a concise concrete syntax.

3. 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 or, barring that, ensure that an appropriately tuned memoization (i.e. caching) strategy is being used.[7]

4. The next problem is that dynamic string parsing mainly decreases the syntactic cost of statically known patterns. Patterns constructed dynamically 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 *)
```

   We needed to use both dynamic string parsing and explicit applications of pattern constructors to achieve the intended semantics. It is difficult to capture idioms like this more concisely using dynamic string parsing (we will see an example of syntax that does capture such idioms below).

5. Finally, for functions like `example` where we are constructing patterns on the basis of data of type `string`, using strings coincidentally to introduce patterns makes it easy for 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 ^ ": \\d\\d\\d-\\d\\d-\\d\\d\\d\\d")
```

[7] Anecdotally, in scientific code using standard contemporary compilers, this optimization is not automatic. Compilation caches are more often provided by the regular expression library, but like any cache, it is difficult to reason compositionally about performance in their presence.

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 today [1]. Proving that mistakes like these have not been made involves reasoning about complex run-time data flows, so it is once again notoriously difficult to automate.

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 [3]. 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 [25]. A formal account of such a system has been developed by Griffin [14]. Unfortunately, these systems are not flexible enough to allow us to express pattern syntax in the conventional manner. For example, sequences of characters can only be parsed as identifiers using these systems, rather than as characters in a regular expression pattern.

Syntax extension systems based on context-free grammars like Sugar* [7], Camlp4 [22] and several others are more expressive, and would allow us to directly introduce 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/
```

For patterns constructed dynamically, 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 much lower syntactic cost, like this:

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

Had we mistakenly written `%name`, we would encounter only a static type error, rather than the silent injection vulnerability discussed above.

All of the problems of dynamic string parsing described above are solved by this approach. Unfortunately, it introduces new problems. First, the systems mentioned thusfar 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 in the same piece of code. So properly considered, every combination of extensions introduced using these mechanisms creates a *de facto* derived syntactic dialect of our language.

In response to this problem, Schwerdfeger and Van Wyk developed a modular analysis that accepts only grammar extensions that specify a universally unique starting token and obey subtle constraints on the follow sets of base language non-terminals [33]. Extensions that satisfy these criteria can be used together in any combination without the possibility of syntactic conflict. However, note that the most natural starting tokens like `pattern` cannot be guaranteed to be universally unique, so we would be forced to use a more verbose token

like `edu_cmu_verse_rx_pattern`. There is no principled way for clients of our extension to define local abbreviations for starting tokens because this mechanism is language-external.

Putting this aside, we must also consider the question of how newly introduced forms elaborate 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 elaboration 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, this issue is not easy to address.

Putting aside this question of hygiene as well, we can address the problem of choosing a module to use in the elaboration by requiring that the client explicitly identify it as an "argument" to the 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 there is no typing discipline – 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 [19] 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 [21]. 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 [20, 13] and integrated into languages, e.g. MacroML [13] and Scala [4].

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}
```

Here, `pattern` is a macro parameterized by a module `P : PATTERN`. It statically parses the provided string literal (which must still be written using an Ocaml-style literal here) to generate an elaboration. The macro specifies that the elaboration will be of type `P.t`. By parsing the string statically, we avoid the problems of dynamic string parsing for statically known patterns.

For patterns that are constructed dynamically, we need to get a bit 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)
```

While this does not leave us with syntax that is quite as distinct and concise as would be possible with a naïve syntax extension, it does avoid many of the problems with that approach: 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 elaboration will not capture variables inadvertently, and using a typed macro system, there is a type discipline (i.e. programmers need not examine the elaboration 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 statically-typed macros. TSMs can be parameterized by modules, so they can be used to define syntax valid at all modules satisfying a specified signature. As we will discuss in 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 leverages 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 resulting in 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, a *parameterized TSM*, `pattern`, is being applied 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 *elaboration*, i.e. another external expression. In this case, `pattern` generates the following elaboration (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(P : PATTERN) at P.t {
  static fn (ps :: ParseStream) :: Exp => (* pattern parser here *)
}
```

This TSM definition first identifies the TSM as `pattern`, then specifies a module parameter, `P`, which must match the signature `PATTERN`. Note that identifying the module parameter as `P` here is an entirely local choice, i.e. the TSM can be applied to *any* module parameter matching `PATTERN`. This parameter is used in the type annotation **`at`** `P.t`, which specifies that all elaborations that arise from the application of this TSM to a module `P` and a delimited form must necessarily be of type `P.t`. These elaborations arise by the static action of the parse function defined next, within braces. It is written as a static function of sort `ParseStream -> Exp`. Both of these are defined in the Verse *prelude*, which is a set of definitions available ambiently. The sort `ParseStream` will give the function access to the body of the delimited form (in blue above) and the sort `Exp` encodes the abstract syntax of external expressions.

To support splicing syntax as described in Sec. 4.2.2, the parse function must be able to extract external subexpressions directly from the parse stream. 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 occur directly in the parse stream. When the parse function encounters these, it asks the `ParseStream` to extract these as *spliced*

expressions for use in the elaboration. For example, the elaboration generated for the body of `example_tsm` above would, if written concretely with spliced expressions marked, be:

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

The hygiene mechanism then ensures that only these marked portions of the generated elaboration can refer to the variables at the use site, preventing inadvertent variable capture by the elaboration. For example, the following would not be a valid elaboration, because it has free variables that are not inside spliced subexpressions:

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

Formalization To briefly 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. `Pattern`, rather than a parameterized family of types):

(syn-aptsm)

$$\frac{\Delta \vdash \psi \mathbin{@} \sigma_{\text{ty}}\ \{\sigma_{\text{parser}}\} \qquad \mathbf{parsestream}(b) = \sigma_{\text{ps}} \qquad \sigma_{\text{parser}}(\sigma_{\text{ps}}) \Downarrow \sigma_{\text{elab}} \qquad \sigma_{\text{elab}} \uparrow e_{\text{elab}} \qquad \Delta; \Gamma; \emptyset; \emptyset \vdash e_{\text{elab}} \Leftarrow \sigma_{\text{ty}} \rightsquigarrow \iota_{\text{trans}}}{\Delta_{\text{out}}; \Gamma_{\text{out}}; \Delta; \Gamma \vdash \mathbf{aptsm}(\psi; b) \Rightarrow \sigma_{\text{ty}} \rightsquigarrow \iota_{\text{trans}}}$$

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. 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.

Note also that we omit various contexts relevant only to other Verse primitives. We include only *typing contexts*, Γ, which map variables to types, and *type abstraction contexts*, Δ, which track universally quantified type variables, both in the standard way. We describe why there are two of each – the *outer contexts* and the *current contexts* – 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} under Δ 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 ParseStream $\to$ Exp), defined by the TSM.

2. The second premise creates a parse stream, σ_{ps}, of sort ParseStream, from the body of the delimited form.

3. The third premise applies the parse function to the parse stream to generate an encoding of the elaboration, σ_{elab}, of sort Exp.

4. The fourth premise decodes σ_{elab}, producing the elaboration itself, e_{elab}.

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

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

   (syn-var)

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

Outer contexts are switched back in when encountering marked spliced expressions (which arise as explained previously from spliced expressions in the parse stream):

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

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 each time. To further lower the syntactic cost of using TSMs, Verse also supports *type-specific languages* (TSLs). This allows library providers to associate a TSM directly with a generated type or parameterized type constructor. For example, a module `P` can associate `pattern` with `P.t` as follows:

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

Local type inference then determines which TSM is implicitly applied when analyzing a delimited form not prefixed by a TSM name. For example, this function is equivalent to `example_tsm` (note the return type annotation):

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

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

```
casetype 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. However, we did not need to build in this syntax primitively.

4.3.3 Limitations

Elaboration validation ensures that an incorrect parser cannot threaten type safety, but proving type safety inductively is subtle because the elaboration is not a subexpression of the expression it will replace. There is a "simple" solution to this problem: the elaboration 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.

Another concern is that if we define the SL such that it supports general recursion, this then causes typechecking to become undecidable, because the parse function may not terminate. In practice, this too should only be a minor concern – a compiler might be expected to limit the running time of the parse function to ensure that compilation does not "hang" due to an incorrect parser. Related concerns would arise if the static language supported state or access to other external effects. Existing techniques for controlling effects would handle this problem, but this is not something we plan to discuss in detail here because our SL does not specify such effects.

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, as it makes maintaining a type discipline in the presence of TSMs substantially easier – neither humans nor tools need to examine the elaboration to determine what type it must necessarily have. We will, however, discuss other points in the design space in the dissertation.

Finally, we have discussed expression syntax here, but have not explicitly 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. We plan to concretely discuss this as the most immediate avenue for future work, but will not consider pattern matching in formal detail.

4.4 Timeline and Milestones

We have described and given a more detailed formal specification of TSMs in a recently published paper [28], and TSLs in a paper published last year [27], both in the context of the Wyvern language. The mechanics of statically invoking the parse function, elaboration encoding/decoding and hygienic elaboration 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 work in the next section to present a unified design. The rule (syn-aptsm) above represents the most substantial step of this plan. I anticipate that this will take about **1.5 weeks** of further work.

Wyvern does not have an ML-style module system (type declarations are the only form of type generation) and at the time, it did not have parameterized types, so to actually support the examples we have given in this section, we also need to update the formalization to handle parameterized TSMs. We plan to flesh this out in the course of writing the dissertation, and I anticipate it requiring 5.5 weeks of my time after the proposal is complete. We propose the following concrete milestones:

1. equip the EL with an appropriately structured module context **(2 weeks)**
2. formally specify the syntax of parameterized TSM expressions **(0.5 weeks)**
3. formally specify an updated version of the TSM expression validation judgement that appeared as the first premise of (syn-aptsm) above **(1 week)**
4. update the metatheory established in the work cited above to handle this **(2 weeks)**

To validate our claims of expressiveness, we plan to give several more examples drawn from existing languages. We 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, we will show how to implement Haskell's primitive "do" syntax using TSLs together with a modular encoding of monads (as described on Prof. Harper's blog [16]). We will also briefly summarize the corpus analysis we conducted in [27] 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. We anticipate that writing these examples up will take an additional **1 week** of work, bringing the total for this section up to 8 weeks.

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 case types (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 polymorphic function types 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, let us define a *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 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. Regular string types classify expressions that behave like strings in the corresponding regular language, here the language of non-empty strings. 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.

We 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`. That 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 and regular strings, 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 both orders *)
  val intro_title_conf : title_t -> conf_t -> t
  val intro_conf_title : conf_t -> title_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 specified their semantics using function types. Expressing labels explicitly is an important facet of having labeled tuple types in a language, so we put them into the function identifiers. Even with our minimal EL, we could implement the semantics of these 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 as well, 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 intro_conf_title conf title => (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 found in Scala [4], adapted to a language with 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. It is not possible to allow clients to omit component labels and rely on the positional information available in the type when introducing a value of record type for this reason.

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

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 TSL 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 (title : title_t, conf : conf_t) => {
  __0_title => title,
  __1_conf => conf
}
fun tpl_of_Paper {__0_title => title, __1_conf => conf} =>
  (title, conf)
```

```
fun Paper_of_rcd {title => title, conf => conf} => {
  __0_title => title,
  __1_conf => conf
}
fun rcd_of_Paper {__0_title => title, __1_conf => conf} = {
  title => title,
  conf => conf
}
```

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 [11].

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* [10]. Such systems typically rely on refinement specifications supplied in 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. We cannot define a function `#group` that can be used 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).

Put another way, type refinements are useful when one needs to make "behavioral" distinctions within a type, but they are not appropriate when new "structural" distinctions are needed. 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, the compiler loses the ability to use a machine representation more suitable to this particular static invariant. Of course, structural distinctions are sometimes inconvenient because they require defining and applying coercions between distinct types, but 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 semantics 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 1 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 1, 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.

The type translator for `Ltuple.c` is defined on lines 20-26 of Figure 1. 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 `list_of_comp_spec :: 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 $\vdash$ `Paper type` $\rightsquigarrow$ `string` $\times$ (`string` $\times$ `unit`). The following rule governs type translation:

(trans-ty)

$$\frac{\vdash_\Phi \mathbf{ty}(m \cdot c; \sigma_{\text{param}}) :: \mathsf{Ty} \qquad \mathbf{ty}(m \cdot c; \sigma_{\text{param}})\ \mathtt{sval}_\Phi \qquad \mathbf{translator}(\Phi; m \cdot c) = \sigma_{\text{translator}} \\ \sigma_{\text{translator}}(\sigma_{\text{param}}) \Downarrow \sigma_{\text{trans}} \qquad \sigma_{\text{trans}} \uparrow^m_\Phi \tau_{\text{abstrans}} \parallel \delta : \Delta \qquad \Delta \vdash \tau_{\text{abstrans}}\ \mathtt{itype}}{\vdash_\Phi \mathbf{ty}(m.c; \sigma_{\text{param}})\ \mathtt{type} \rightsquigarrow [\delta]\tau_{\text{abstrans}}}$$

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 $\mathbf{ty}(m \cdot c; \sigma_{\text{param}})$, 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 $\alpha_1 \times (\alpha_2 \times \texttt{unit})$. 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 1, 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` like this, because the return type of the function determines the expected type:

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

First, we provide 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 1, `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(conf : conf_t, title : title_t) : Paper =>
  ltpl Ltuple Paper_tyidx {conf, title}
```

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(conf : conf_t, title : title_t) : Paper =>
  {conf, title}
```

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 1. 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.

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).[8] 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

[8]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.

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), ()))‘
```

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 under a context where $x_0 : \alpha_0$ and $x_1 : \alpha_1$ against the abstracted type translation for the type the operation as a whole was being analyzed against, i.e. $\alpha_0 \times (\alpha_1 \times \mathsf{unit})$, as discussed above. Only after this succeeds are the actual argument translations substituted in to generate the final operation translation, here $(conf, (title, ()))$.

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. If the regular string metamodule later decides to change how it implements regular string types, this would require only local changes to that metamodule. We call this property *translation independence*, by analogy to the *representation independence* property that underlies modular reasoning with ML modules. An important point is that 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 would need to introduce a number of technical devices. It would 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.

The remaining opcons are synthetic opcons, which means they can be used anywhere. They operate similarly, but rather than taking a type as input, they produce a pair of a type and a translation as output. In every other way, these are analagous to analytic opcons, so we omit the details here. Instead, 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 exceptions, 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 typing contexts as an avenue for future work in the thesis.

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 a rule. 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, case types, 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 derived concrete syntax and 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) => fold (list_of_comp_spec 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)) => tycase(ty) {
      c(ty_param) => (fold (zipExact (list_of_comp_spec 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)) =>
      (* ... same as intro_unlabeled but reorder first ... *)
  }

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

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

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

Figure 1: Using metamodules to define the type structure of labeled tuple types.

References

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

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

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

[4] E. Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 3:1–3:10, New York, NY, USA, 2013. ACM.

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

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

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

[8] 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*. 2013.

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

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

[11] 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 2014)*. ACM, 2014.

[12] M. Furr and J. S. Foster. Checking type safety of foreign function calls. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 62–72, New York, NY, USA, 2005. ACM.

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

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

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

[16] R. Harper. Of Course ML Has Monads! `https://existentialtype.wordpress.com/2011/05/01/of-course-ml-has-monads/`.

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

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

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

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

[21] E. E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Symposium on LISP and Functional Programming*, pages 151–161, Aug. 1986. To appear in Lisp and Symbolic Computation.

[22] 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.

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

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

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

[26] 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.

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

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

[29] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active Code Completion. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 859–869, Piscataway, NJ, USA, 2012. IEEE Press.

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

[31] J. C. Reynolds. GEDANKEN - a simple typless language based on the principle of completeness and reference concept. *Comm. A.C.M.*, 13(5), May 1970.

[32] 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.

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

[34] E. Spishak, W. Dietl, and M. D. Ernst. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, pages 20–26. ACM, 2012.

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

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

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