Modularly Metaprogrammable Syntax and Type Structure (Thesis Proposal)

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

Abstract

Functional programming languages like ML descend conceptually from 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 in this design space, as evidenced by the diversity of dialects that continue to proliferate around major languages. But distinct language dialects cannot be modularly composed, limiting the choices ultimately 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 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 their design. For example, they build in record types, generalizing the nullary and binary product types more common in minimal calculi, because labeled components are often cognitively useful. Similarly, they build in derived concrete syntax for commonly used constructs like lists.

Ideally, only a limited number of "embellishments" like these would be needed to turn a minimal lambda calculus into a practical "full-scale" functional programming language. The community around the resulting language would then be able to direct its efforts almost exclusively toward the development of useful libraries, optimizing compilers and other tools. Unfortunately, a stable language design that fully achieves this ideal has yet to emerge, as evidenced by the diverse array of *dialects* – languages derived from a common base language that introduce different embelishments – that continue to proliferate around all major contemporary languages. In fact, tools that assist with the construction of so-called "domain-specific" language dialects seem only to be increasingly popular. This calls for an investigation: why are programmers and researchers 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 functional languages?

Perhaps the most common reason 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 are often tempted to 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 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 similarly benefits 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. Though we will be able to express precisely the semantics that we seek, the usual approach of using dynamic string parsing to introduce a pattern will leave something to be desired syntactically. Tools like Camlp4 [18] and Sugar* [6, 7] are designed to lower the engineering costs of constructing derived syntactic dialects in situations like these, contributing to their continued proliferation.

More advanced dialects define new static type structure, going beyond what is possible by context-free elaboration to the existing language. Record types themselves are perhaps the simplest example – they cannot be expressed by context-free elaboration to a language with only nullary and binary products. Various dialects of ML and Haskell 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 one example in Sec. 5). The OCaml dialect of ML primitively builds in "polymorphic variants", "open datatypes" and logic for typechecking operations that use format strings like `sprintf` [18]. ReactiveML builds in primitives for functional reactive programming [20]. ML5 builds in high-level primitives for distributed programming based on a modal lambda calculus [22]. Manticore [8] and Alice ML [28] 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 [10]. Tools like proof assistants and logical frameworks can be used to specify and reason metatheoretically about dialects like these, and tools like compiler generators 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 [23]). Second, primitives introduced earlier in a language's lifespan end up monopolizing finite "syntactic resources", forcing subsequent primitives to use ever more esoteric forms. A third problem is that primitives 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 [27]:

> *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 [33]. Using GADTs, Ocaml was able to move some of the *ad hoc* machinery for typechecking operations that use format strings, like `sprintf`, out of the language and into a library (note however that syntactic machinery remains built in).

2 Proposed Contributions

Our 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 derived concrete syntax and type structure of a variety of designs in a safe and modularly composable manner.

Verse features a module system taken directly from SML. Unlike SML, the Verse core language is split into a *typed external language* (EL) specified by type-directed translation to a minimal *typed internal language* (IL). We formally introduce the semantics of the Verse core language in Sec. 3. Our novel contributions will be two primitive constructs that eliminate the need for many others:

- **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 over the parsing and elaboration of delimited segments of concrete syntax.
- **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 consider by giving library providers static hooks directly into Verse's type-directed translation semantics. For example, a library provider can implement the type structure of records by defining a metamodule that:
  1. introduces a type constructor, `record`, parameterized by finite mappings from labels to types, and defines its semantics by translation to unary and binary products (which are built in to the internal language); and
  2. introduces operators used to work with records, minimally record introduction and elimination (but perhaps also various functional update operators), and directly defines the logic governing their typechecking and translation to the IL (which builds in only nullary and binary products).

  We will see direct analogies between modules and metamodules in Sec. 5.

[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.

Both TSMs and metamodules involve *static metaprogramming*, i.e. their static semantics will involve the evaluation of user-defined functions that manipulate static representations of types and terms.

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 a language's 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 semantics of these primitives.[2] Our main technical contributions will come in showing how to address these problems in a principled manner. In particular, syntactic 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]

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 metaprogrammatically express new derived syntax and external type structure atop a minimal typed internal language while maintaining a hygienic type discipline and modular reasoning principles.

2.2 Disclaimers

Before we continue, let us 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 typeclasses, which also have the potential for "abuse").

Finally, we will not be interested here in languages that feature full-spectrum dependent types, which blur the phase separation between "compile-time" and "run-time", though we conjecture that the primitives we describe could be added to languages like Gallina (the "external language" of the Coq proof assistant) with some suitable modifications. Verse is designed for use in situations where ML, Haskell or Scala would be suitable today.

3 Overview of Judgements

To situate ourselves within a formal framework, let us begin with a brief overview of how Verse is organized and specified.

[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 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 [13, 19] (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 the formal details.

3.2 Core Language

The underlying core language – the language of types and expressions – will be the focus of our formal efforts. The Verse core language is organized much like the first stage of a type-directed compiler (e.g. the TIL compiler for Standard ML [31]), consisting of a user-facing *typed external language* (EL) specified by type-directed translation to a minimal *typed internal language* (IL).

3.2.1 Internal Language

Let us begin with the internal language. For our purposes, we keep things simple by using the strictly evaluated polymorphic lambda calculus with nullary and binary product and sum types and recursive types. We assume here that the reader has familiarity with these (we follow *PFPL* [14] directly). To briefly review, the main judgements in the static semantics of the IL take the following standard form (omitting contexts for now):

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

The dynamic semantics can be 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`.

Features like state, exceptions, minimal concurrency primitives, scalars, arrays and others characteristic of a first-stage compiler intermediate language would also be included in the IL in practice, and in some of these cases, this would affect the shape of the internal semantics. However, our design is largely insensitive to the details of the internal language – the only strict requirements are that the IL be type safe and support parametric type abstraction – so we will stick to this simpler core here.

3.2.2 External Language

Programs evaluate as internal expressions but they are written as external expressions. For concision, we will use the terms "type" and "expression" without qualification to refer to *external types*, σ, and *external expressions*, e, respectively. Note that external types are distinct from internal types. 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.

Note 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") [26]. This is to allow us to explicitly specify how Verse's *local type inference* works, and in particular, how it interacts with the primitives that we aim to introduce. Like Scala, we intentionally do not attempt to support global type inference.

4 Modularly Metaprogrammable Textual Syntax

Verse specifies a textual concrete syntax for the EL.[4] Because the purpose of concrete syntax is to serve as a programmer-facing user interface, 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 most functional languages, so that instead of having to write out `Cons(1, Cons(2, Cons(3, Nil)))`, the programmer can equivalently write `[1, 2, 3]`. Many languages go beyond this, building in derived syntax associated with various other types of data, like vectors (the SML/NJ dialect of SML), arrays (Ocaml), monadic commands (Haskell), syntax trees (Scala), XML documents (Scala, Ur/Web) and SQL queries (F#, Ur/Web).

Verse takes a less *ad hoc* and more modular 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.

We will begin in Sec. 4.1 by considering in detail a representative example for which such a mechanism might be useful: regular expression patterns expressed using abstract data types. In Sec. 4.2, we will demonstrate that the usual approach of using dynamic string parsing to introduce patterns is not ideal, and then survey existing alternatives, finding that they involve an unacceptable loss of modularity and other undesirable trade-offs. In Sec. 4.3, we outline our proposed mechanisms and discuss how they will resolve these issues. We conclude in Sec. 4.4 with a timeline for remaining work.

4.1 Motivating Example: Regular Expression Syntax

Let us begin by taking the perspective of a regular expression library provider. Recall that regular expressions are a common way to capture patterns in strings [32]. 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 [14]. 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

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

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 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)
}
```

Any client of a module `P` that satisfies `PATTERN`, written `P : PATTERN`, can manipulate patterns as terms of type `P.t` using the interface described by this signature. By holding the representation type of patterns abstract, the burden of proving that the case analysis function cannot be used by clients to distinguish patterns with the same normal form is localized to each module implementing this signature. The details are standard and not particularly relevant for our purposes, so we omit them here.

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

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

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 parse function that transforms strings to patterns. Because, as just mentioned, there may be many implementations of the `PATTERN` interface, the standard approach is to define a parameterized module (a.k.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 the 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 programmer would see an error message like `error: illegal escape character`.[5] In a small lab study, we observed that this sort of error was common, and that diagnosing the problem nearly always required a non-trivial amount of time for even experienced programmers who had not used regular expressions recently [25]. One workaround with higher syntactic cost is to double 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 allows the following, which is perhaps more acceptable:

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

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

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

   Such problems can sometimes be discovered via testing, but 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" [30].

   We might instead attempt to treat the well-formedness of patterns constructed from strings as a static verification condition. Automatically proving that this condition holds throughout a program is difficult to do in general, because it involves reasoning about arbitrary dynamic behavior. Here, the decision procedure must know that the variable `pattern` is equal to the function `PU.parse`. If the argument had not been written literally but rather computed, e.g. if we had written `"(G" ^ "C"` where `^` is the string concatenation function applied in infix style, we would 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 harder still. Asking the client to provide a proof of well-formedness would defeat the purpose of providing a concise concrete syntax.

3. Parsing patterns dynamically also incurs a performance cost. Regular expression patterns are often used on large datasets, so it is easy to 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 transformation or ensure that an appropriately tuned caching strategy is being used.[6]

4. The next problem is that dynamic string parsing only decreases the cost of statically known patterns, but not those constructed using other values. For example, consider this function from strings to patterns:

[5] 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`.

[6] Anecdotally, based on my experience in bioinformatics, the most common situation is that the compiler is not generally willing to partially evaluate expensive computations like pattern parsing out of such expressions automatically, and the programmer is often unaware of the performance cost.

```
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 more concise syntax below).

5. Finally, for functions like the one just described where we are constructing patterns on the basis of data of type `string`, it is easy for programmers to make mistakes using string concatenation. For example, consider the following "alternative" definition of `example`:

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

Note that both functions 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 `name` contains special characters that have meaning in the concrete syntax of patterns that a problem arises. In applications that query sensitive data, problems like this lead to *injection attacks*, which are common and can be catastrophic from the perspective of software security [1]. Proving that mistakes like these have not been made involves reasoning about complex run-time data flows, so it is once again difficult to automate.

The difficulties 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), such problems can arise. This 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 approach is to use a system that gives library providers the power to directly extend the concrete syntax of our language with new derived syntactic 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 [21]. A formal account of such a system has been developed by Griffin [12]. 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 [18] 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/
```

Unfortunately, this approach is perilous because 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 [29]. 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, there is also 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 has not been addressed.

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/
```

For patterns constructed compositionally, we can define *splicing* syntax for both strings (using prefix `@`) and other patterns (using prefix `%`). For example, we can write `example_bad` instead like this:

```
fun example_fixed1(name : string) =>
  edu_cmu_verse_rx_pattern P /@name: %ssn/
```

Had we mistakenly used the pattern splicing syntax, writing `%name`, rather than the string splicing syntax, `@name`, we would encounter only a static type error, rather than the silent injection vulnerability that we discussed earlier.

A general problem with approaches like these (in addition to those already mentioned) is that they suffer from a paucity of direct reasoning principles – given an unfamiliar piece of syntax, there is no straightforward method for determining what type it will have, or even for identifying which extension determines its elaboration, 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 [15] is the most prominent example of such a system. Naïvely, this system also suffers from the problem of unhygienic variable capture, but modifications of this system, notably in the Scheme dialect of LISP, support reasoning hygienically about the rewriting that a macro performs [17]. In languages with a stronger static type discipline, variations on macros that restrict rewriting to a particular type and perform the rewriting statically have also been studied [16, 11] and integrated into languages, e.g. MacroML [11] and Scala [4].

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

```
let 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, specified by the macro to be at type `P.t`.

For patterns that are constructed compositionally, we would need to get more creative. For example, we might repurpose the infix operators that the language normally uses 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 clean 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), 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 by using a typed macro system, there is a typing discipline (i.e. we need not examine the elaboration directly to know what type it must have).

4.3 Contributions

We will now introduce a new primitive – the **typed syntax macro** (TSM) – that combines the syntactic flexibility of syntax extensions with the reasoning guarantees of statically-typed macros, and show how to integrate it with the Verse module system. This will address all of the problems brought up in the previous two subsections. We will introduce TSMs in Sec. 4.3.1.

We then go further in Sec. 4.3.2, introducing **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, further decreasing syntactic cost (which is, of course, the entire purpose of this exercise).

The result is that derived syntax defined by library providers using TSMs and TSLs provides comes at syntactic cost comparable to that of derived syntax built in primitively by a language designer or using a naïve tool like Camlp4, without the associated problems.

4.3.1 Typed Syntax Macros (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/` (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 the parameterized TSM `pattern` has the following basic form:

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

We first identify the TSM, then specify the module parameter. Note that identifying it as `P` here is not important, i.e. the TSM can be used with *any* module matching signature `PATTERN`. The module parameter is used in the type annotation **`at`** `P.t`, which specifies that all elaborations successfully generated by applying this TSM to a module parameter `P` and a delimited form will be of type `P.t`. Elaborations are generated by the static action of the parse function defined on the second line. This parse function must be a static value of type `ParseStream -> Exp`. Both of these types are defined in the Verse *prelude*, which is a set of definitions available ambiently. The type `ParseStream` will give the function access to the body of the delimited form (in blue above) and the type `Exp` is a case type that

encodes the abstract syntax of external expressions. In other words, this function parses the body of the delimited form to generate an encoding of the elaboration.

The parse function can identify portions of the body of the delimited form that should be treated as spliced expressions, allowing us to support splicing syntax as described in Sec. 4.2.2:

```
(* 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/
```

A hygiene mechanism ensures that only portions of the generated elaboration derived from such spliced expressions can refer to the variables at the use site, preventing inadvertent variable capture by the elaboration. Put another way, the elaboration logic must be valid in any typing context.

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

(syn-aptsm)

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

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

1. The first premise can be pronounced "TSM expression s is a TSM at type σ under Δ with parser ι_{parser}". This judgement would handle the mechanisms of module parameter application, but we must elide the details here (a complete specification of this judgement represents the main piece of remaining work).

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

3. The third premise applies the parser to the parse stream to generate an encoding of the elaboration, ι_{elab}. This is where the metaprogramming occurs.

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

5. The fifth premise validates the elaboration by analyzing it against the type σ under empty contexts. The *current* typing and type abstraction contexts are "saved" for use when a spliced term is encountered during this process by setting them as the new *outer contexts*.

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

   (syn-spliced)

   $$\frac{\emptyset; \emptyset; \Delta_{\text{out}}; \Gamma_{\text{out}} \vdash e \Rightarrow \sigma \rightsquigarrow \iota}{\Delta_{\text{out}}; \Gamma_{\text{out}}; \Delta; \Gamma \vdash \mathbf{spliced}(e) \Rightarrow \sigma \rightsquigarrow \iota}$$

   (ana-spliced)

   $$\frac{\emptyset; \emptyset; \Delta_{\text{out}}; \Gamma_{\text{out}} \vdash e \Leftarrow \sigma \rightsquigarrow \iota}{\Delta_{\text{out}}; \Gamma_{\text{out}}; \Delta; \Gamma \vdash \mathbf{spliced}(e) \Leftarrow \sigma \rightsquigarrow \iota}$$

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

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

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

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

4.3.2 Type-Specific Languages (TSLs)

To further lower the syntactic cost of using TSMs, Verse also supports *type-specific languages* (TSLs), which allow library providers to associate a TSM directly with a generated type. 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 applied when analyzing a delimited form not prefixed by a TSM name. For example, this is equivalent to `example_tsm`:

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

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

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

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

```
let 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.

4.4 Timeline

We have described and given a more detailed formal specification of TSMs in a recently published paper [24], and TSLs in a paper last year [23], both in the context of the Wyvern language. The basic mechanics of elaboration encoding/decoding and hygienic elaboration validation have all been satisfyingly specified in this context.

Wyvern does not specify an ML-style module system (type declarations provide the only form of type generativity), so in the context of Verse, there are some extensions that need to be defined in the formalization to handle module parameters. We plan to complete this in the course of writing the dissertation, and anticipate it requiring about 3 weeks of effort over Summer 2015. We emphasize again that we do not plan to specify the full module system in detail; we will assume that judgements of an appropriate form have been specified and detail only those portions that interface with the novel aspects of the core language.

5 Modularly Metaprogrammable Type Structure

Let us now turn our attention to the type structure of the Verse EL. In contrast to other languages, which build in constructs like case types (shown briefly in use in the previous

section), tuples, records, objects and so on primitively, in Verse these can all be expressed as modes of use of *metamodules* (which are distinct from, though conceptually related to, modules, as we will see). Only polymorphic function types are built primitively into the Verse EL.

This section is organized much like Sec. 4. We begin with examples of type structure that we wish 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 consider a type classifying conference papers:

```
let type Paper = ltuplex {
   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*) `ltuplex` to a statically valued ordered finite mapping from labels to types, here written using a conventional concrete syntax within curly braces (this is a mode of use of TSLs, as we will see). Its first component is labeled `title` and specifies the regular string type `rstring /.+/`. Regular string types are constructed by applying the tycon `rstring` to a regular expression pattern, again written using a conventional concrete syntax. Regular string types classify values 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 static label, written literally here as `<conf>`. The resulting *operator*, `# <conf>`, takes one *argument*, `exmpl`.

Given a value of regular string type like this, we can project out captured groups using the `#group` operator constructor, which is parameterized by a 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]+/`.

Labeled tuples support a number of other basic operations. For example, labeled tuples can be concatenated (with any common components updated with the value on the right) using the `ltuple+` operator:

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

5.2 Existing Approaches

Before considering labeled tuples and regular strings as a mode of use of metamodules, let us consider some alternative approaches.

5.2.1 Labeled Tuple Types

Recall that Verse builds in only nullary and binary products into its internal language, so the type structure described above for labeled tuples is not directly expressible using IL primitives. The EL does not build in even these. However, 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 One approach we might take is to attempt to express the type structure of any particular labeled tuple type using the module system. For our example type `Paper` from above, we 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
}
```

We've simply taken all possible valid introductory and projection operators and called for their expression as functions, moving label parameters into 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). Alternatively, if we slightly relax our insistence on minimalism and add binary products to the EL, we could use them to implement these operators as well:

```
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
}
```

However, there are several problems with this basic 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. For example, 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).

Another issue is that we can only reasonably express the introductory and projection operators by enumerating them in this way. To enumerate all possible operators that arise from the opcon `ltuple-` as functions would require constructing an encoding of every labeled tuple type that might arise from dropping one or more components. This would

add an exponential factor to our volume of boilerplate code. Finally, there is simply no way to define `ltuple+` as a finite collection of functions because there are infinitely many extensions of any labeled tuple type.

Records and Tuples If, instead of settling for such a module-based encoding, we further weaken our insistence on minimalism and primitively build in record/tuple types, as SML and many other languages have done, let us consider what might be possible. We note at the outset that records and tuples 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 the same component mapping:

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

Unlike labeled tuple types, record types are identified only up to component reordering, so this mapping does not preserve certain type disequalities between labeled tuple types. This means 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 this type.

To work around this limitation if we do not otherwise 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 run-time costs of explicitly invoking these functions. We could define a TSL for every such type to reduce this cost to clients, though this is an additional cost to providers.

If on the other hand we do care about preserving the type disequalities in question, 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 this 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 again bear the cost of applying these conversion functions ahead of every operation and providers again bear the burden of defining this boilerplate code (which again has volume linear in the number of components, albeit with a constant factor that is again not easy to dismiss). And once again, the provider must verify that these functions were implemented correctly.

Finally, note that neither of these workarounds provides us with any way to uniformly express the more interesting operations, like concatenation or dropping a component, that we discussed earlier.

5.2.2 Regular Strings

Let us now turn to regular string types.

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

Type Refinements We might instead try to express the static semantics of regular strings by using standard strings and moving the logic governing regular string types into an external system of *type refinements* [9]. Such systems typically rely on refinement specifications supplied in comments:

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

One problem with this is that there is 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 this function would need information that is available only in the type refinement of each regular string, i.e. the positions of the captured groups. 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)
```

Ensuring that there will not be an out-of-bounds error here would again require more sophisticated extrinsic reasoning.

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. Of course, for regular strings, there may be limited benefit (or even some cost) to controlling representation differentially, but in other situations, this could have significant performance ramifications. For example, it is quite common to justify the omission of a primitive type of "non-negative integers representable in 32 bits" by the fact that this is "merely" a refinement of (mathematical) integers, but in so doing the compiler loses the ability to use a machine representation more suitable to this particular static invariant.

Modules To create type disequalities, we might consider again turning to the module system, defining a signature for each regular string type that we wish to work with. Again, this requires a non-trivial amount of boilerplate code:

```
signature TITLE = sig {
  type t
  val intro : string -> t
```

```
  val strcase : t -> (unit -> 'a) -> (string * string -> 'a) -> 'a
}

signature CONF = sig {
  type t
  val intro : string -> t
  val strcase : t -> (unit -> 'a) -> (string * string -> 'a) -> 'a
  val prj_group_0 : t -> string
  val prj_group_1 : t -> string
}
```

To ensure that applications of the introductory operators are correct statically, we would need to couple this with a system of type refinements. Moreover, this again reveals itself to be too limited a solution when we need to express other operators. For example, expressing concatenation uniformly using a single function or a finite set of functions is impossible, for the same reason as applied to `ltuple+` above, i.e. there are infinitely many regular string types that might be involved.

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 the type structure of labeled tuples and regular strings more directly, as we will show below.

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*. Briefly summarized:

- A *metasignature* specifies a set of type and operator constructors (tycons and opcons). Each is parameterized by a specified *kind* of *static value*.
- A *metamodule* defines static functions that govern the typechecking and translation to the IL of each type and operator constructor specified in the metasignature.

Static values are values of the *typed static language* (SL), which itself forms a total typed lambda calculus, with *static expressions* σ classified by *kinds* κ. Types are one kind of static value. We discuss other kinds of static values below.

5.3.1 Example: Labeled Tuple Types via Metamodules

An example of a metasignature, `LTUPLEX`, and a matching metamodule, `Ltuplex`, is shown in Figure 1. We also show a *metamodule-parameterized TSM* `ltplx` to decrease syntactic cost. The remainder of this subsection explains these definitions.

Type Constructors On line 2, the metasignature `LTUPLEX` specifies a type constructor `c` parameterized by static values of kind `LblTyOrdMap`. This user-defined kind (not shown here for brevity) classifies ordered mappings from labels (of kind `Lbl`, also user-defined but not shown) to types, which are static values of kind `Ty` (a primitive kind). Let us assume that we have defined a *kind-specific language* for `LblTyOrdMap`, so we can write a static value of this kind concretely like this:

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

We can construct a type by applying the type constructor `Ltuplex.c` to this parameter:

```
let type Paper = Ltuplex.c Paper_ty_idx
```

```
metasignature LTUPLEX = metasig {
  tycon c of LblTyOrdMap
  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 ltplx(L :: LTUPLEX, param :: LblTyOrdMap) at L.c(param) {
  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 Ltuplex :: LTUPLEX = metamod {
  (* translate labeled tuple types to nested products *)
  tycon c {
    (* type translation generator: *)
    fn (param :: LblTyOrdMap) => fold (list_of_ltom param)
      `unit`
      (fn ((lbl, ty), cur_ty_trans) => `trans(ty) * %cur_ty_trans`)
  }

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

  syn opcon intro_labeled {
    fn (param :: list(Lbl), args :: list(Arg)) =>
      (* ... same as intro_unlabeled but reorder first ... *)
  }

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

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

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

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

Line 58 defines the tycon synonym `ltuplex` for `Ltuplex.c`. If we also substitute in the value of `Paper_ty_idx` just given, we see that we have recovered the definition of `Paper` given at the beginning of Sec. 5.1:

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

Note that because types are static values, this is equivalent to writing:

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

Recall from Sec. 3 that each valid external type has a translation, which is an internal type. The metamodule `Ltuplex` defines a *type translation schema*, a static function that determines the translations of types constructed by `c` (each identified by the value of its parameter). Internal types are encoded as static values of kind `ITy`, which we assume supports standard *quasiquotation* syntax. Here, the type translation schema on lines 19 to 25 generates nested internal products by folding over a list generated from the type parameter (in the standard way), returning the encoding of `unit` for the empty case, and recursively nesting encodings of binary products otherwise using the standard *unquote* operator, `%`. For example, the encoding 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 that we plan to explore in the remaining work.

If, for example, all regular string types translate to standard internal strings, encoded by some internal type abbreviated `string`, and substitute this in, then the translation of `Paper` is ultimately:

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

Operator Constructors On line 3 of Figure 1, the metasignature `LTUPLEX` specifies an operator constructor, `intro_unlabeled`, parameterized by static values of kind `unit`. The qualifier **ana** specifies that this opcon can only be used in analytic positions (i.e. where the expected type is known). For example, `Ltuplex.intro_unlabeled` can be applied like this, because the return type of the function determines the expected type:

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

First, a static parameter value of the appropriate kind is supplied. After that, an *argument list*, `(conf; title)`, is supplied.

The metamodule `Ltuplex` determines how this expression is typechecked and translated on lines 27-36. Analytic opcons like `intro_unlabeled` are defined with a static function that is given 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 provided arguments. The function must return an encoding of an internal expression, which will become the translation of the external expression. Internal expressions are encoded as static values of kind `IExp`, and again we assume that we can use standard quasiquote syntax. Here, we simply generate the trivial value for an empty labeled tuple and recursively generate nested pairs otherwise, following directly the structure for the type translation generator above. This is, of course, intentional: the introductory operation for a type must generate an expression translation consistent with the corresponding type translation for type safety to hold.

describe ltplx?

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. For concision, we omit the details. Instead, note that the code written in these definitions is essentially exactly what would appear in a standard implementation of the first stage of a compiler. The novelty is not in the definitions, but in how they are organized and brought into the language.

5.4 Timeline

For the sake of concision, we leave providing substantially more technical details on how metamodules operate as work that remains to be done. However, we have a lengthy draft of a technical report that provides these details (for an older version of this work). We anticipate completing this work will take about 2 months during Summer 2015.

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] 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.

[9] 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.

[10] 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.

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

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

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

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

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

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

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

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

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

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

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

[22] 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.

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

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

[25] 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.

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

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

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

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