Modularly Metaprogrammable Syntax and Type Structure (Thesis Proposal)

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

Abstract

Functional programming languages like ML descend conceptually from simple lambda calculi, but to be pragmatic, must expose a syntax and type structure to programmers of a decidedly more elaborate design. The designers of languages like these have many viable choices, as evidenced by the diverse array of dialects that continue to proliferate around all major contemporary languages. Unfortunately, language dialects cannot, in general, be modularly composed, limiting the choices that are available to programmers. We propose a new functional language, Verse, designed around novel metaprogramming mechanisms that 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 typed lambda calculus called the *internal language*.

1 Motivation

Functional programming languages like Standard ML (SML) and Haskell descend directly from elegant typed lambda calculi, diverging mainly in that they consider various human factors in addition to fundamental metatheoretic issues in their design. For example, they both build in record types, generalizing the nullary and binary product types that are more common in simpler calculi, because labeled components are cognitively useful. Similarly, they both build in derived syntax for the standard list datatype.

Ideally, the transition from a minimal calculus to a pragmatic “full-scale” programming language would require only a limited number of primitive “embellishments” like these. The community around the language could then direct its efforts almost exclusively toward the development of useful libraries. Unfortunately, a single language design that achieves this ideal has proven elusive, as evidenced by the diverse array of dialects that continue to proliferate around all major contemporary languages, functional languages included. In fact, tools that make constructing “domain-specific” language dialects easier appear to be increasingly popular. This calls for an investigation: why are well-informed programmers and researchers so often unable to satisfyingly express the constructs that they need in libraries, as modes of use of the “general-purpose” functional primitives already available in major contemporary languages today?

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. For example, we consider regular expression patterns expressed using abstract data types (in ML, a mode of use of the module system) in Sec. 4. This gives us precisely the semantics that we seek to express, but syntactically, we argue that the usual approach of using strings to introduce patterns leaves much to be desired. To avoid this, we are tempted to introduce derived syntax specific to our library by constructing a *derived syntactic dialect*, i.e. a dialect specified by context-free elaboration to the existing language. This is far from an isolated example. For example, Ur/Web is a derived syntactic dialect of Ur (itself descended from ML) that builds in derived syntax for SQL queries, HTML elements and other datatypes common in web programming [5]. Tools like Camlp4 [17] and Sugar* [6, 7] (which we will return to later) are designed to lower the engineering costs of constructing derived syntactic dialects, contributing to their proliferation.

More advanced dialects introduce new type structure, going beyond what is possible by context-free elaboration to an existing language. For example, as mentioned above, SML goes beyond nullary and binary product types to expose record types. Other dialects have explored "record-like" primitives that support functional update and extension operators, width and depth coercions (sometimes implicit), mutable components, methods, prototypic dispatch and various other embellishments. Ocaml builds in an object system, polymorphic variants and conveniences like logic for typechecking operations that use format strings, e.g. `sprintf` [17]. ReactiveML builds in primitives for functional reactive programming [19]. ML5 builds in primitives for distributed programming [21]. Manticore builds in parallel arrays [8]. Alice ML builds in futures and promises [27]. MLj builds in essentially the entire Java programming language, to support typesafe interoperation with existing Java programs [2]. Indeed, perusal of the proceedings of any major programming languages conference will typically yield several more examples. Tools like proof assistants and logical frameworks support specifying and reasoning metatheoretically about dialects like these, and tools like compiler generators and language workbenches 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 program written in one dialect cannot, in general, safely and idiomatically interface with libraries written in a different dialect. To do so would fundamentally require combining the two dialects into a single language. Left unconstrained, this is an ill-posed problem, i.e. given two dialects that are specified by judgements of arbitrary form, it is not clear what it would even mean to combine them. Even when the dialects are specified using some formalism that does provide a means to naïvely combine two dialects, there is generally no guarantee that the composition will conserve important syntactic and semantic properties. For example, consider two derived syntactic dialects, one building in derived syntax for JSON (a popular data interchange format), the other using a similar syntax for finite mappings of some other type. Though each is known to have an unambiguous concrete syntax in isolation, when their syntax is naïvely combined (e.g. by Camlp4), ambiguities arise. Any putative "combined language" must formally be considered a distinct system for which one must derive all metatheorems of interest anew, guided only informally by those derived for the dialects individually. Due to this paucity of modular reasoning principles, language dialects are not practical artifacts for software development "in the large".

Dialects do have an indirect impact on large-scale software development because the language designers that control comparatively popular languages, like Ocaml and Scala, can be convinced to incorporate ideas from dialects into backwards compatible language revisions. However, this has come at a cost: these languages have ballooned in size. And yet, because there appears to be a "long tail" of potentially useful primitives (for derived syntax, our group has gathered initial data speaking to this [22]), primitives that are only situationally useful, or that trigger aesthetic disagreements between different factions of the community, are still quite often left languishing in "toy" dialects. Recalling the words of Reynolds, which are as relevant today as they were nearly half a century ago [26]:

> *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 language design community with two possible paths forward if we wish to keep general-purpose languages small and free of *ad hoc* primitives. One, exemplified (arguably) by SML, is to generally eschew the introduction of new primitives and settle on a set of primitives that sit at a "sweet spot" in the overall language design space, accepting that in some circumstances, this trades away expressive power or leads to high syntactic cost. The other path forward is to continue on in the search for even more general language primitives, i.e. primitives that are so expressive that they permit us to degrade a broad class of existing primitives, including those found in dialects, to modularly composable library constructs, so that they can be evaluated on their individual merits by programmers and used together without the possibility of conflict. Encouragingly, primitives that have such a "slimming" effect on a language do occasionally arise. For example, Ocaml in a recent revision added support for "generalized algebraic data types" (GADTs), based on research on guarded recursive datatype constructors [32]. Using GADTs, Ocaml was able to move some of the machinery for typechecking operations that use format strings, like `sprintf`, out of the language and into the standard library (unfortunately, some machinery remains built in). Our broad aim is to introduce a new functional programming language called Verse[1] that takes even more dramatic steps down this second path.

2 Proposed Contributions

Verse has a module system taken directly from SML, but its core language is specified by a *type-directed translation semantics* targeting a minimal typed lambda calculus, called the Verse *internal language*, introduced in Sec. 3. In lieu of typical *external* (i.e. programmer-facing) derived syntax and type structure, Verse introduces two novel primitive constructs:

- **Typed syntax macros** (TSMs), introduced in Sec. 4, subsume 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, subsume the need to primitively build in the type structure of constructs like records (and variants thereof) by giving library providers static hooks directly into the type-directed translation semantics governing the Verse external language. For example, we will discuss how a library provider can define a single metamodule that defines *labeled tuple types* together with the operators that allow clients to introduce, project out of and concatenate values of these types. The internal language builds in only nullary and binary product types primitively.

Both TSMs and metamodules can be understood as *metaprogramming* primitives, because they involve programmatic static code generation.

The key challenge in the design of these primitives will come in ensuring that they are metatheoretically well-behaved, given that they aim to give library providers decentralized control over aspects of Verse's concrete syntax and type structure that, in other languages, are under the monolithic control of the language designer. If we are not careful, many of the problems discussed earlier as inherent to combining distinct language dialects 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 made impossible by construction and we will validate the code statically generated by TSMs and metamodules to maintain a *hygienic type discipline* and, most uniquely, powerful *modular reasoning principles*. In other words, library providers will

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

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

have the ability to reason about the constructs that they define in isolation, and clients will be able to use them safely in any combination, without the possibility of conflict.[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, 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 typeclasses, which also have the potential for "abuse").

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

3 Summary of Judgements

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

The Verse module language is based directly on SML's module language, with which we assume a working familiarity for the purposes of this proposal [12, 18] (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.

The core language is where Verse diverges more radically from other languages, in that it is organized much like the first stage of a type-directed compiler (e.g. the TIL compiler for Standard ML [30]), consisting of a user-facing *typed external language* (EL) specified by type-directed translation to a minimal *typed internal language* (IL). The main judgements in the specification of the EL take the following form (omitting various contexts for now):[4]

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

Note that the expression typing judgements are *bidirectional*, i.e. we make a judgemental

[3] We assume that naming conflicts can be avoided extrinsically, e.g. by using a URI-based naming scheme as in the Java ecosystem.

[4] Note that we use the terms "expression" and "type" without qualification to refer to external expressions and types, respectively.

distinction between *type synthesis* (the type is an "output") and *type analysis* (the type is an "input") [25]. 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, primarily because it is not technically possible to do so (but also because we believe that compilers for languages that support only local type inference generate clearer error messages).

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. For our purposes, we will keep things simple by using the strictly evaluated polymorphic lambda calculus with nullary and binary product and sum types and recursive types, which we assume the reader has familiarity with and for which all the necessary metatheorems are well-established (we follow [13]). The main judgements in the static semantics of the IL (again omitting contexts for now) take the following standard form:

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, simple 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 cases this would affect the shape of the internal semantics, but we omit them for simplicity.

4 Modularly Metaprogrammable Textual Syntax

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

Rather than privileging particular library constructs with primitive syntactic support, Verse exposes primitives that allow library providers to introduce new derived syntax on their own. To specifically motivate our desire for this level of syntactic control, we begin in Sec. 4.1 with a representative example: regular expression patterns expressed using abstract data types. We show how the usual approach of using strings to introduce patterns is not ideal. We then survey existing alternatives in Sec. 4.2, finding that they involve an unacceptable loss of modularity and other undesirable trade-offs. In Sec. 4.3, we outline our proposed mechanisms and discuss how they 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 from the perspective of a regular expression library provider. Recall that regular expressions are a common way to capture patterns in strings [31]. We will assume that the abstract syntax of patterns, p, over strings, s, is specified as below:

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

The most direct way to express this abstract syntax is by defining a recursive sum type [13]. Verse supports these as case types, which are analagous to datatypes in ML (we will see how the type structure of case types can in fact be expressed in libraries later):

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

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

```
signature PATTERN = 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)
end
```

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

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

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

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

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

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

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

to construct patterns like this:

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

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 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 solving the problem was nearly always initially challenging to even experienced programmers who hadn't used regular expressions recently [24]. The inelegant workaround is to double backslashes:

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

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

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

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

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

   Such issues can sometimes be found via testing, but empirical data gathered from open source projects suggests that there remain many malformed regular expression patterns that are not detected by a project's test suite "in the wild" [29].

   We might instead attempt to treat the well-formedness of patterns constructed from strings as a static verification condition. Statically proving that this condition holds throughout a program is wildly difficult to automate in general, because it involves reasoning about arbitrary dynamic behavior. In the example above, we must know that the variable `pattern` binds the function `PU.pattern`. 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 the infix style, we would need to be able to establish that this expression is equivalent to the string above. For patterns that are dynamically constructed based on input to the program (further discussed below), the problem is harder still.

[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`.

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

3. The final problem is that using strings to construct patterns encourages programmers to use operations over strings, like concatenation, to construct patterns derived from user input that happens also to be of type `string`. This can lead to subtle but serious bugs. For example, consider the following function:

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

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

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

   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), so testing can miss the problem. Proving that such problems cannot arise involves reasoning about complex run-time data flows, so it is once again difficult to automate. In applications that query sensitive data, mistakes like this lead to *injection attacks*, which are both common and catastrophic from the perspective of software security [1].

4.2 Existing Approaches

The difficulties described above are common whenever strings are parsed dynamically to address the issue of syntactic cost (a static issue). This has motivated a substantial quantity of research on reducing this use of run-time string parsing [3]. The existing approaches can be classified in one of two ways: as being based on *syntax extension* or *static term rewriting*.

4.2.1 Syntax Extension

One approach we might consider is to directly extend the syntax of our language with new derived syntactic forms.

The simplest such syntax extension systems are those where 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 [20]. A formal account of such a system has been developed by Griffin [11]. 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 pattern. Syntax extension systems based on context-free grammars like Sugar* [7], Camlp4 [17] and many others are more expressive, and would allow us to directly introduce pattern syntax into our core language's grammar, perhaps like this:

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

However, this is a rather perilous approach because none of these systems guarantee freedom from *syntactic conflicts*, i.e. as the Coq manual states: "mixing different symbolic notations in [the] same text may cause serious parsing ambiguity". So if another library provider used similar syntax for a different implementation of regular expressions, or for some other unrelated abstraction, then a client could not simultaneously use both libraries in the same scope. Properly considered, every combination of extensions introduced using one of 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 [28]. Extensions that satisfy these criteria can be used together in any combination without the possibility of 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 now decide how our newly introduced forms elaborate to forms in our base grammar. 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` satisying `PATTERN` is in scope is a brittle solution. In fact, we should expect that the extension mechanism actively prevents such capture of specific variable names to ensure that variables (including module variables) can be freely renamed. Such a *hygiene discipline* is well-understood only when performing term-to-term rewriting (discussed below) or in simple rewrite systems like those found in Coq. For more flexible mechanisms, the issue is a topic of ongoing research (none of the grammar-based mechanisms described above enforce hygiene).

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

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

For patterns constructed compositionally, we can define *splicing* syntax for both strings and other patterns, e.g. as shown below:

```
fun example_syn(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 simple 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.

4.2.2 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*. Among the earliest examples is the LISP macro system [14], later modified, notably in the Scheme language, to support reasoning hygienically about the rewriting that is performed [16]. In languages with a stronger static type discipline, variations on macros that restrict rewriting to a particular type and perform the rewriting in an explicitly staged manner have also been studied [15, 10] and integrated into languages, e.g. MacroML [10] 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 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 address many of the problems we discussed above: 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 propose a new primitive, the **typed syntax macro** (TSM), that combines the flexibility of syntax extensions with the reasoning guarantees of statically-typed macros, and show how to integrate it into a language with an ML-style module system, addressing all of the brought up thusfar. We then introduce **type-specific languages** (TSLs), which leverage local type inference to control TSM 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 approaches the cost of derived syntax built in primitively by a language designer (perhaps using a naïve tool like Camlp4).

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 applied to a module parameter, `P`, and a delimited form, `/A|T|G|C/`. 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, it is as if we wrote the following directly:

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

The definition of `pattern` is shown below:

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

Note that identifying the module parameter as `P` here is unimportant, 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` will be of type `P.t`. Elaborations are generated by the static action of the parse function defined in the indented block on the second line. The parse function must be of type `ParseStream -> Exp`. Both types are defined in the Verse *prelude*, which is a set of definitions available ambiently. The type `ParseStream` gives the function access to the body of the delimited form (in blue above) and the type `Exp` is a case type that encodes the abstract syntax of external expressions.

For concision in analytic positions, i.e. those where an expected type is known due to, for example, a type ascription, the module parameter `P` can be inferred:

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

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

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

TSMs can 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. 4.2.1:

```
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 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)

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

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 avenue for 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}.

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*.

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

   (syn-var)

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

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

   (syn-spliced)

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

   (ana-spliced)

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

   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)))
   ```

4.3.2 Type-Specific Languages (TSLs)

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

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

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 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.

4.4 Timeline

We have described and given a more detailed formal specification of TSMs in a recently published paper [23], and TSLs in a paper last year [22], both in the context of the Wyvern language. The mechanics of elaboration generation, encoding and 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 a form of type generativity), so in the context of Verse, there are some changes that need to be made in the formalization to handle module parameters, as mentioned earlier. We plan to complete this in the course of writing the dissertation, and do not anticipate it requiring more than 1-2 weeks of effort over the summer.

5 Modularly Metaprogrammable Type Structure

Let us now turn our attention to the type structure of the Verse external language (EL). In contrast to other languages, which build in constructs like case types (shown briefly at the beginning of Sec. 4.1), 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 Section 4. We begin with examples of type structure that we wish to express in Section 5.1, then discuss how existing approaches are insufficient in Section 5.2. Next, we show how metamodules (together with TSMs and TSLs) serve to address these problems in Section **??** and conclude with a timeline in Section **??**.

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 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 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 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 exmpl_conf_name = #group 0 exmpl_conf
```

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

Our take on labeled tuples supports 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 a : ltuple {authors : list(rstring /.+/)} = {["Harry Q. Bovik"]}
let 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 exmpl_paper_anon = ltuple- <authors> exmpl_paper_final
```

5.2 Existing 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 example, we might define the following signature for our example type Paper from earlier:

```
signature PAPER
  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 expressed them as functions, moving label parameters into identifiers. Even with our minimal EL, we could implement this signature using a Church-style encoding (the details are standard and omitted). Alternatively, if we only slightly relax our insistence on minimalism and add binary products to the EL, we could use these as well:

```
module Paper : PAPER
  type t = title_t * conf_t
  fun intro_tpl 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

type Paper = Paper.t
```

There are several basic problems with this approach. First, each labeled tuple type that a library provider wishes to encode in this way requires a volume of boilerplate code factorial in the number of components it has (because we must consider all permutations of label orderings as a distinct introductory function). It is also not satisfying syntactically from the client's perspective. To avoid these problems and recover the syntax used previously, we could define a TSM for every such signature, though this too involves much boilerplate.

More fundamentally, we could only express the introductory and projection operators. We were not able to enumerate all possible operators that arise from the opcon ltuple- as functions, because to do so, we would need to construct an encoding of every labeled tuple type that might arise from dropping one or more components. This would require an exponential volume of boilerplate code. Finally, there is simply no way to define ltuple+ as a finite collection of functions – there are infinitely many extensions of any particular labeled product 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 defining 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. In other words, this mapping does not preserve certain type disequalities between labeled tuple types. Lacking positional information, it is not possible to omit component labels and rely on positional information when introducing a value of this type directly.

To work around this limitation if we do not otherwise care about preserving these type disequalities, we would need to define two conversion functions involving 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 providers, this has syntactic cost linear in the number of components. For clients, we must explicitly invoke these functions, which incurs both a syntactic and run-time cost. We could define a TSM 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 (which is again linear in the number of components, albeit with a constant factor that is again not easy to dismiss).

As with our modular encoding above, neither of these workarounds provides us with any way to uniformly express the more interesting operations, like concatenation, 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 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 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.

A closely related problem is that type refinements do not introduce type disequalities. As a result, compilers cannot use the type refinements to optimize the representation of a value. For example, the fact that some value of type `string` can be given the refined type `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. For regular strings, there may be limited benefit to controlling representation differentially, but in other situations, this can 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 integers, but in so doing one 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
  type t
  val intro : string -> t
  val strcase : t -> (unit -> 'a) -> (string * string -> 'a) -> 'a

signature CONF
  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 refinement types.

Unfortunately, this again reveals itself to be too limited a solution when we need other operators, e.g. concatenation. Expressing concatenation uniformly using a single function or a finite set of functions is impossible, for the same reason that applied to `ltuple+` above, i.e. there are infinitely many regular string types.

5.3 Contributions

Metamodules introduce type and operator constructors parameterized by arbitrary *static values* and metaprogrammatically express the logic governing their typechecking and translation to the IL. *Metasignatures* classify metamodules. Let us begin by considering the following metasignature, which classifies metamodules that express the semantics of labeled tuples as described above:

On line 2, we declare a tycon, identifying it as `c`, and specify that it is parameterized by static values of type `LblTyOrdMap`, which we assume has been defined as the type of ordered finite mappings from labels, which are values of type `Lbl`, to types, which can be manipulated as static values of type `Ty`.

```
metasignature LTUPLEX
  tycon c of LblTyOrdMap
  ana opcon intro_unlabeled (*of unit*)
  ana opcon intro_labeled of list(Lbl)
  syn opcon # of Lbl
  syn opcon + (*of unit*)
  syn opcon - of Lbl

syntax ltpl(L ~ LTUPLEX, param : LblTyOrdMap) at L.c(param)
  fn ps => (* ... intro syntax ... *)

metamodule Ltuplex ~ LTUPLEX with syntax ltpl(Ltuplex) at c
  (* translate labeled tuple types to nested products *)
  tycon c(param : LblTyOrdMap) ~> (fold (list_of_ltom param)
    ‘unit‘
    (fn ((lbl, ty), cur_trans) => ‘trans(ty) * %cur_trans‘)
  )

  (* translate intro operator to nested pairs *)
  ana opcon intro_unlabeled(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)}, %cur_trans)‘)
      )
      _ => raise TyErr("...")
    }
  )

  (* same as intro_unlabeled but reorder first *)
  ana opcon intro_labeled(ty : Ty, param : list(Lbl), args : list(Arg)) ~>
    (* ... *)

  (* generate the appropriate n-fold projection out of nested pairs *)
  syn opcon #(param : Lbl, args : list(Arg)) ~>
    (* ... *)

  (* generate the new type ..... *)
  syn opcon +(param : unit, args : list(Arg)) ~>
    (* ... *)
  syn opcon -(param : Lbl, args : list(Arg)) ~>
    (* ... *)
 (* 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.

Let us assume a metamodule `LTupleX` that implements this metasignature,
c is a tycon parameterized by static maps from labels to types.
intro unlabeled is a trivially parameterized opcon, i.e. just one operator.
intro labeled is an opcon parameterized by a list of labels.
# is an opcon parameterized by a single label.
ltpl is a TSM parameterized by a metamodule and a static value
LTuple uses ltpl as a TSL

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] 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.

[11] 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.

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

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

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

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

[16] 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.

[17] 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.

[18] 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.

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

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

[21] 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.

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

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

[24] 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.

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

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

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

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