Modularly Metaprogrammable Syntax and Type Structure (Thesis Proposal)

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

Abstract

We propose a new general-purpose programming language called Verse. Like ML, Verse consists of a module language atop a core language of types and expressions. The Verse module language is taken directly from ML. The core language consists of an external language (EL) specified by type-directed translation to a minimal internal language (IL). Uniquely, the EL is also remarkably minimal. In lieu of typical syntax and type structure, Verse introduces novel metaprogramming mechanisms that give library providers the ability to modularly express concrete syntax and type structure of many designs. For example, library providers can express list syntax, HTML syntax, regular expression pattern syntax, tuples/records, extensible tuples/records, labeled sums, object systems, operations on format strings like `sprintf` and typechecked foreign function interfaces.

These mechanisms integrate cleanly with Verse's module system and come equipped with modular reasoning principles of their own: library providers can prove a broad class of metatheorems about the constructs they have defined in a "closed world" where no other constructs need to be considered. A translation validation process ensures that all such metatheorems will necessarily continue to hold no matter which other constructs are also being used in a program, i.e. in the "open world". As in the module system, type abstraction plays a critical role in this process.

1 Introduction

Ideally, a general-purpose programming language should only rarely need to be extended or forked to form a dialect. Instead, programmers should be able to express the constructs that they need in libraries, as modes of use of existing constructs, and reason about their behavior modularly. Languages like ML, which features a powerful module system atop a core type structure rooted in the foundations of logic, appear to have put this ideal well within reach today. Yet despite their expressive power, new dialects of languages like ML, and tools for creating such dialects, do continue to arise. What is motivating this?

Perhaps the most common motivation is a concern about the *syntactic cost* of expressing a construct of interest using general-purpose mechanisms (often assessed qualitatively [13], though quantitative metrics can be defined). For example, we consider regular expression patterns expressed using abstract data types (in ML, a mode of use of the module system) in Sec. 3. There is no semantic issue with this approach, but syntactically, it leaves much to be desired. In response to situations like this, it is not uncommon for providers to consider a *syntactic dialect* of the language, i.e. one that specifies new concrete syntax by context-free elaboration to the existing language. For example, Ur/Web is a dialect of Ur (itself descended from ML) that builds in concrete syntax for SQL queries, HTML documents and other datatypes common in web programming [6]. Indeed, nearly all languages include some derived syntax for constructs that are otherwise expressed in the standard library, e.g. list syntax in Standard ML. Tools like Camlp4 [17] and Sugar* [7, 8] are designed to lower the engineering costs of constructing syntactic dialects.

More advanced dialects build in constructs that cannot simply be defined by a context-free elaboration to an existing language. For example, dialects of ML variously provide "record-like" constructs that support functional update and extension operators, mutable rows, width and depth coercions (often implicit) [5], methods (i.e. function members that receive a self-reference implicitly), prototypic dispatch and various other embellishments. Ocaml builds in a class-based object system as well as various conveniences, e.g. logic for typechecking operations that use format strings, like `sprintf`. ReactiveML builds in constructs for functional reactive programming [18]. ML5 builds in constructs for distributed programming [20]. Manticore builds in parallel arrays [9]. Alice ML builds in futures and promises . MLj builds in essentially the entire Java programming language, citation to support typesafe interoperation with existing Java code [3]. Indeed, perusal of the proceedings of any major programming languages conference will typically yield several more examples. Tools like proof assistants and logical frameworks are designed to support specifying and reasoning about dialects like these, and tools like compiler generators and language workbenches simplify the task of implementing them.

The problem with this practice of using language dialects as vehicles for new syntactic and semantic constructs is that it is, in an important sense, anti-modular: a program written in one dialect cannot, in general, safely and idiomatically interface with libraries written in a different dialect. This would require constructing a language where the constructs of both dialects are identically expressible, and where all the metatheoretic properties relevant to reasoning about program behavior are conserved. There can be no guarantee that such a language exists in general. As a very simple example, consider two 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. Each is known to have an unambiguous concrete syntax in isolation, but when their syntax is naïvely combined, ambiguities arise. Combining the semantics of two languages, particularly when they are specified in very different ways, is even more of an art than a science, and subject to often subtle failures.

Given this, there are two approaches that the language design community can consider to put the ideas in dialects like those above into the hands of software engineers. One, exemplified by languages like Scala and Ocaml, is to progressively borrow new ideas from dialects as they emerge, integrating them into backwards compatible revisions. In taking this approach, power as well as responsibility to maintain metatheoretic guarantees is concentrated around a small group of language designers. This results in a large core language, yet because of a "long tail" effect, constructs that are situationally useful (or where there is disagreement) can still end up marginalized. Moreover, constructs that are included but turn out to be flawed remain entrenched (and monopolize finite "syntactic resources", as we will discuss below).

Verse aims to explore the polar opposite approach: a core language that builds in only minimal syntax and type structure *a priori*. Instead, the core language is organized around metaprogramming mechanisms that give library providers the ability to introduce new concrete syntax and type structure of many designs, while imposing enough structure to ensure that the language maintains strong safety and modularity-related guarantees. This is far from a trivial matter. For example, if we simply conceived of the core language's semantics as being specified by a "bag of rules" (like "paper" specifications of languages often appear) and let library providers introduce new rules, one could easily define rules that violated type safety, interfered with invariants being maintained by rules defined in other libraries, or introduced non-determinism. There is also the question of how to allocate finite syntactic resources. For example, as noted above, the design space around record types is large. If we allow libraries to explore this design space, which library gets to determine the meaning of a form like `{label1=e1, label2=e2}`?

Before we discuss how we address these problems, it is useful to 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. For example, constructs that require that the type system support subtyping could not be introduced uncompromisingly into a language that has no such semantic notion. Our interest is only in a subset of all constructs that can be specified in a mutually orthogonal manner by a semantics of a certain prototypic "shape". We will make this more specific below.

2 Contributions

2.1 Verse

To provide a coherent vehicle for our contributions, we introduce a new programming language called Verse. Let us briefly review its organization. Syntactically, Verse has a layout-sensitive textual concrete syntax (i.e. newlines and indentation are meaningful). This choice is not fundamental to our proposed mechanisms, but it will be useful for cleanly expressing a class of examples that we will discuss later. Semantically, Verse is organized into a *module language* and a *core language*. The module language is based directly on the Standard ML module language, with which we assume familiarity for the purposes of this proposal [15]. We will use it in an example in Section 3. The core language, i.e. the language of types and expressions, is split into an *external language* (EL) and an *internal language* (IL). The EL is specified by a type-directed translation semantics targeting the much simpler IL. Notionally, this can be thought of as simply shifting the first stage of a type-directed compiler directly into the language specification [26]. More specifically, we specify a bidirectionally typed translation semantics, i.e. one where a distinction is made between *type analysis* (the type is an "input") and *type synthesis* (the type is an "output"), for reasons that we will return to later.

The mechanisms we introduce are relatively insensitive to the particular choice of IL. Choosing the semantics of the IL is the primary decision that a language designer leveraging these mechanisms would need to make. The only strict requirements are that the IL is type safe and supports parametric type abstraction. For our purposes, we will keep things simple by using the strictly evaluated polymorphic lambda calculus with nullary and binary product and sum types and recursive types, which we assume the reader has familiarity with and for which all the necessary metatheorems are already well-established [16]. Features like state, exceptions, concurrency primitives, scalars, arrays and other constructs characteristic of a first-stage compiler IL would also be included in practice, but we omit them when their inclusion would not meaningfully affect our discussion.

2.2 Modularly Metaprogrammable Concrete Syntax

There are three main mechanisms that we propose in this thesis. The first two have to do with concrete syntax, and are closely related, while the final one has to do with the type structure of the core language.

- **Typed Syntax Macros**. TODO: description + example (and a variant of them that further lowers syntactic cost in certain circumstances, *type-specific languages*), introduced in Sec. 3.
- **Tycon Structures**, introduced in Sec. 4.

For each mechanism that we introduce, we will first demonstrate its expressive power by showing how constructs that are, or would need to be, built directly in to contemporary languages can in Verse be expressed in libraries. To demonstrate that these mechanisms are theoretically sound, we will then develop a formal specification and proofs of various important metatheoretic properties. The most interesting of these will be various *modular* *reasoning principles* that guarantee that user-defined constructs can be reasoned about separately and then used together in any combination, without the possibility of syntactic or semantic conflict.

2.3 Thesis Statement

In summary, we propose a thesis defending the following statement:

> A programming language specified by a type-directed translation semantics can give library providers the ability to introduce derived concrete syntax and new external type structure, housed in constructs that can be reasoned about separately and used together in any combination.

3 Modular Derived Syntax

Most contemporary computer programming languages specify a textual concrete syntax. Because it serves as the language's human-facing user interface, it is common practice to include derived syntactic forms (colloquially, *syntactic sugar*) that capture common idioms more concisely or naturally. For example, 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 other types of values, like vectors (SML), arrays (Ocaml), commands (Haskell) and syntax trees (Scala).

Rather than privileging particular data structures, Verse instead exposes mechanisms that allow library providers to introduce new derived syntactic constructs on their own. To motivate our desire for this level of syntactic control, we begin in Sec. 3.1 with the example of regular expression patterns expressed using abstract types, showing how the usual workaround of using strings to introduce patterns is not ideal. We then survey existing syntax extension mechanisms in Sec. 3.2, finding that they involve an unacceptable loss of modularity and other undesirable trade-offs. In Sec. 3.3, we outline our proposed mechanisms and discuss how they resolve these issues. We conclude in Sec. 3.4 with a timeline for remaining work.

3.1 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 [27]. 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 [16]. Verse supports these as case types, which are analagous to datatypes in ML:

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

However, there are some reasons not to expose this representation of patterns directly to clients. First, patterns are usually identified up to congruence. For example, $\textbf{seq}(\textbf{empty}, p)$ is congruent to p. It would be useful if congruent patterns were 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 here) without exposing this "implementation detail" to clients. Indeed, there are many ways to implement regular expression patterns, each with different performance trade-offs. For these reasons, a better approach is to define the following *module type* (a.k.a. *signature* in SML), 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`. Notice that it exposes an interface otherwise isomorphic to the one available using a case type:

```
module type PATTERN
  type t
  val Empty : t
  val Str : string -> t
  val Seq : t * t -> t
  val Or : t * t -> t
  val Star : t -> t
  val Group : t -> t
  val case : (
    'a ->
    (string -> 'a) ->
    (t * t -> 'a) ->
    (t * t -> 'a) ->
    (t -> 'a) ->
    (t -> 'a) ->
    'a)
```

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 encoding:

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

To encode the concrete syntax of patterns, regular expression library providers usually provide a utility function that converts strings to patterns. Because there may be many pattern implementations, the usual approach is to define a parameterized module (a.k.a. *functor* in SML) defining such utility functions generically, like this:

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

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

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

to construct patterns like this:

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

This approach is imperfect for several reasons:

1. Our first problem is syntactic (or, one might say, aesthetic): string escape sequences conflict syntactically with pattern escape sequences. For example, the following will not be well-formed:

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

   In fact, when compiling an analagous term using SML of New Jersey (SML/NJ), we encounter the rather puzzling error message `Error: unclosed string`. Using `javac`, we encounter `error: illegal escape character`. In a small lab study, we observed that this error was common, and nearly always initially misinterpreted by even experienced programmers who hadn't used regular expressions recently [22]. The workaround is to double backslashes:

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

2. Our second problem has an impact on both correctness and performance: pattern parsing does not occur until the pattern is evaluated. For example, the following malformed pattern will only trigger an error when this term is evaluated during the full moon:

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

   Such issues can be found via testing, but empirical data gathered from open source projects suggests that there are many malformed regular expression patterns that are not detected by a project's test suite "in the wild" [25].

   Parsing patterns at run-time also incurs a performance penalty. To avoid incurring it every time the pattern is encountered during evaluation, an appropriately tuned caching strategy must be introduced, increasing client complexity.

3. The final problem is that using strings to introduce patterns makes it more likely that programmers will use string concatenation to construct patterns derived from other patterns or from user input. For example, consider the following function:

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

   The (unstated) intent here is to treat `name` as a pattern matching only itself, but this is not the observed behavior when `name` contains special characters that are meaningful in patterns. The correct code is more verbose, again resembling abstract syntax:

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

   The mistake was the result of a programmer using a flawed heuristic, so it could be avoided with sufficient developer discipline. The problem is that it is difficult to enforce this discipline mechanically. Both functions above have the same type and behave identically at many inputs, particularly those that would be expected during typical executions of the program (i.e. alphabetic names). In applications that query sensitive data, failures of this variety are observed to be both common and catastrophic from the perspective of security [2]. Ideally, our library would be able to make it more difficult to inadvertently introduce subtle security bugs like this.

3.2 Existing Approaches

These problems with string-oriented approaches to concrete syntax arise in many similar scenarios [21], motivating research on more direct syntax extension mechanisms [4].

The simplest such mechanisms are those where each new syntactic form is described by a single equation. For example, the surface language of Coq (called Gallina) includes such a mechanism [19]. A theoretical account of such mechanisms has been developed by Griffin [14]. Unfortunately, these mechanisms are not able to express pattern syntax in the conventional manner for various reasons. For example, sequences of characters should not be parsed as identifiers.

Syntax extension mechanisms based on context-free grammars like Sugar* [8], Camlp4 [17] and many others are more expressive, and would allow us to directly introduce pattern syntax into our base language's grammar. However, this is perilous because none of the mechanisms described thusfar guarantee *syntactic composability*, i.e. as stated 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 variant of regular expressions, or for an entirely unrelated abstraction, then a client could not simultaneously use both libraries in the same scope.

In response to this problem, Schwerdfeger and Van Wyk have developed a modular analysis that accepts only syntax extensions that use a unique starting token and satisfy some subtle conditions on follow sets of base language non-terminals [24]. However, simple starting tokens like `pattern` cannot be guaranteed to be globally unique, so we would need to use a more verbose token like `edu_cmu_wyvern_rx_pattern`. There is no simple, principled way to define scoped abbreviations for starting tokens because this mechanism is language-external.

In any case, we must now decide how our newly introduced derived forms desugar to forms in our base language, which in this case requires determining which module the constructors the desugaring uses will come from. Clearly, simply assuming that a module named `P` satisfying `PATTERN` is in scope is a brittle solution. Indeed, we should expect that the extension mechanism actively prevents such capture of specific variable names to ensure that variables (including module variables) can be freely renamed. However, we note that such *hygiene mechanisms* are only well-understood when performing term-to-term rewriting (as in Lisp-style *macro systems*) or simple equational 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).

citation

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

```
let N = edu_cmu_wyvern_rx_pattern[P] /A|T|G|C/
```

Syntactically, this is the best we can do using existing approaches.

These approaches further suffer from a paucity of direct reasoning principles, i.e. the program can only be reasoned about post-desugaring. Given an unfamiliar piece of syntax, there is no simple method for determining what type it will have, or even for identifying which extension determines its desugaring, causing difficulties for both humans (related to code comprehension) and tools.

3.3 Contributions

We propose designing a syntax extension mechanism that addresses all of the problems just described. It consists of two constituent concepts:

- The concept that the mechanism is oriented around is the *typed syntax macro* (TSM). For example, consider the following concrete term:

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

  The TSM `pattern` is being applied to a module parameter, `P`, and a *delimited form*, `/A|T|G|C/`. This term elaborates *statically* to the following:

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

  The TSM is defined as shown below:

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

  We declare a module parameter (calling it `P` here is unimportant, i.e. the TSM can be used with *any* module satisfying module type `PATTERN`), and the type clause **`for`** `P.t`, which guarantees that any use of this TSM will either be ill-typed or elaborate to a term of type `P.t`. The elaboration is computed by the compile-time action of the parse function defined in the indented block above. This function must be of type `ParseStream -> Exp`, where the type `ParseStream` gives the function access to the *body* of the delimited form (in blue above) and the type `Exp` encodes the abstract syntax of the base language. Both types are defined in the Verse prelude, which is a set of definitions available ambiently.

  When the type of the term is known, e.g. due to a type annotation on the let binding, the module parameter `P` can be inferred:

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

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

```
let syntax pat = pattern[P]
pat /A|T|G|C/
```

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

```
let N = pat /A|T|G|C/
let BisI = pat /GC({N})GC/
```

A hygiene mechanism ensures that only those portions of the elaboration derived from such spliced terms can refer to variables in the surrounding scope.

Strings are never involved, so the attendant security issues described previously are easily avoidable. That is, the following

- To support an even more concise usage profile, Verse also supports *type-specific languages* (TSLs), which allow library providers to associate a TSM directly with a declared type. For example, the module `P` above can associate `pattern` with `P.t`. Local type inference then determines which TSM is applied implicitly to handle a form not prefixed by a TSM name. For example, the following is equivalent to the above:

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

3.3.1 Typed Syntax Macros (TSMs)

For simplicitly, let us begin with a typed syntax macro providing pattern syntax using the case type encoding above:

```
syntax pattern => Pattern = e_parser
```

The term `e_parser`, elided here, must have type `ParseStream -> Exp`, where `ParseStream` classifies a sequence of characters and `Exp` classifies a reified Verse expression.

3.3.2 Type-Specific Languages (TSLs)

3.4 Timeline

SAC, ECOOP.
TODO: module-parameterized TSMs

4 Extensible Semantics

As a minimal example, consider System **F**, a.k.a. the polymorphic lambda calculus (perhaps the simplest general-purpose language) [12, 23, 16]. It specifies type variables and two type constructors, written abstractly `arr` and `all`:

Sort			Abstract Form	Typeset Form	Description
Type	τ	$::=$	t	t	type variable
			$\texttt{arr}(\tau;\tau)$	$\tau \rightarrow \tau$	function type
			$\texttt{all}(t.\tau)$	$\forall t.\tau$	type quantification

and value variables and four term constructors:

Sort			Abstract Form	Typeset Form	Description
Exp	ι	$::=$	x	x	value variable
			$\texttt{lam}[\tau](x.\iota)$	$\lambda x{:}\tau.\iota$	value abstraction
			$\texttt{ap}(\iota;\iota)$	$\iota(\iota)$	application
			$\texttt{Lam}(t.\iota)$	$\Lambda t.\iota$	type abstraction
			$\texttt{Ap}[\tau](\iota)$	$\iota[\tau]$	type application

If we wanted to express record types in **F**, derived syntax would not be enough (the static semantics of **F** provide no way to express row labels, amongst other issues). We could, however, develop a dialect that built in the type and term operators relevant to records, specifying its static and dynamic semantics as a type-directed translation to the polymorphic lambda calculus (using the standard Church-style encoding for records). More formally, assuming that metavariables e, σ and Υ range over terms, types and typing contexts of our dialect, and ι, τ and Γ range over the terms, types and typing contexts of **F**, the main judgements in the specification of our dialect would take the following form:

Judgement	Pronunciation
$\Upsilon \vdash e : \sigma \rightsquigarrow \iota$	Under typing context Υ, term e has type σ and translation ι.
$\Upsilon \vdash \sigma \rightsquigarrow \tau$	Under typing context Υ, type σ is valid and has translation τ.
$\vdash \Upsilon \rightsquigarrow \Gamma$	Typing context Υ is valid and has translation Γ.

couple more sentences

4.0.1 Example 2: Regular Strings

The example above deals with how regular expression patterns are encoded, introduced and reasoned about. Going further, programmers may also benefit from a semantics where they can statically constrain strings to be within a particular regular language [11]. For example, a programmer might want to ensure that the arguments to a function that creates a database connection given a username and password are alphanumeric strings, ideally including this specification directly in the type of the function:

```
type alphanumeric = rstring /[A-Za-z0-9]+/
val connect : (alphanumeric * alphanumeric) -> DBConnection
```

This example requires that the language include a type constructor, `rstring`, indexed by a statically known regular expression, written using the syntax described above. Ideally, we would be able to use standard string literal syntax for such regular strings as well, e.g.

```
let connection = connect("admin", "password")
```

To be useful, the language would also need a static semantics for standard operations on regular strings. For example, concatenating two alphanumeric strings should result in an alphanumeric string. This should not introduce additional run-time cost as compared to the corresponding operations on standard strings. Coercions that can be determined statically to be valid in all cases due to a language inclusion relationship should have trivial cost, e.g. coercions from alphabetic to alphanumeric strings.

Regular strings might go beyond simply *refining* standard string types (i.e. specifying verification conditions atop an existing semantics [10]). For example, they might define an operation like *captured group projection* that has no analog over standard strings:

```
let example : rstring /(\d\d\d)-(\d\d\d\d)/ = "555-5555"
let group0 (* : rstring /\d\d\d/ *) = example#0
```

It should be possible to give this operation a cost of $\mathcal{O}(1)$.

4.0.2 Example 3: Labeled Products with Functional Update Operators

The simplest way to specify the semantics of product types is to specify only nullary and binary products [16]. However, in practice, many more general but also more complex variations on product types are built in to various dialects of ML and other languages, e.g. n-ary tuples, labeled tuples, records (identified up to reordering), records with width and depth coercions [5] and records with functional update operators (also called *extensible records*) [17]. The Haskell wiki notes that "extensible records are not implemented in GHC. The problem is that the record design space is large, and seems to lack local optima. [...] As a result, nothing much happens." [1]

We would ideally like to avoid needing the language designer to decide *a priori* on just one privileged point in this large design space. Instead, the language designer might define only nullary and binary products, and include an extension mechanism that makes it possible for these other variations on products to be defined as libraries and used together without conflict. For example, we would like to define the semantics of labeled products, which have labeled rows like records but maintain a row ordering like tuples, in a library. An example of a labeled product type (constructed by the type constructor `lprod`) classifying conference papers might be:

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

The row ordering should make it possible to introduce values of this type either with or without explicit labels, e.g.

```
fun make_paper(t : rstring /.+/) : Paper = {title=t, conf="EXMPL 2015"}
```

should be equivalent to

```
fun make_paper(t : rstring /.+/) : Paper = (t, "EXMPL 2015")
```

(note that we aim to be able to unambiguously re-use standard record and tuple syntax.)

We should then be able to project out rows by providing a positional index or a label:

```
let test_paper = make_paper "Test Paper"
test_paper#0
test_paper#conf
```

We might also want our labeled tuples to support functional update operators. For example, an operator that dropped a row might be used like this:

```
let title_only (* : lprod {title : rstring /.+/} *) = test_paper.drop[conf]
```

An operation that added a row, or updated an existing row, might be used like this:

```
fun with_author(p : Paper, a : rstring /.+/) = p.ext(author=a)
```

An important point, however, is that records, as well as all of these "record-like" constructs, can be expressed by a *type-directed translation* targeting an intermediate language that builds in only functions, nullary and binary products, recursive types and reference cells [16] (and indeed, many compilers take advantage of facts like this to simplify the number of cases that must be considered downstream). Substantially more sophisticated examples require at most only a very slightly more capable intermediate language (the IL of popular virtual machines, even those with flaws like ubiquitous `null` references, e.g. JVM or CIL bytecode, are also sufficient if used strictly as translation targets).

References

[1] GHC/FAQ. `http://www.haskell.org/haskellwiki/GHC:FAQ#Extensible_Records`.

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

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

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

[5] L. Cardelli. A semantics of multiple inheritance. In *Proc. Of the International Symposium on Semantics of Data Types*, pages 51–67, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

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

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

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

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

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

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

[12] J.-Y. Girard. Une extension de l'interpretation de gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. *Studies in Logic and the Foundations of Mathematics*, 63:63–92, 1971.

[13] T. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

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

[15] R. Harper. Programming in standard ml, 1997.

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

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

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

[20] 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.

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

[22] 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.

[23] J. C. Reynolds. An introduction to polymorphic lambda calculus. In *Logical Foundations of Functional Programming*, pages 77–86. Addison-Wesley, 1994.

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

[25] 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.

[26] 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.

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