Modularly Defining Derived Syntactic and Semantic Constructs (Thesis Proposal)

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

Abstract

We propose a thesis defending the following statement:

> A programming language can give library providers the ability to introduce derived concrete syntax and type and term operators in a manner that ensures these constructs can be reasoned about separately and composed arbitrarily.

1 Introduction

Well-designed general-purpose programming languages are remarkably stable: they rarely need to be revised or forked into dialects, because programmers can express the constructs that they need just as well in orthogonal libraries. Abstraction mechanisms informed by type theory, like the ML module system [14], have put this ideal within reach. Yet despite their expressive power, new dialects of languages like ML continue to arise. Why is this?

Perhaps the most common justification is that the *syntactic cost* of a desirable construct is not preserved when it is expressed using general-purpose mechanisms. For example, the semantics of regular expressions can be expressed using abstract data types (which in ML are a mode of use of the module system), but as we will detail in Sec. 3, this presents some syntactic difficulties. For programmers in problem domains where regular expressions are common, e.g. bioinformatics, it is tempting to design a "domain-specific" dialect that differs only in that it includes derived syntax (i.e. "syntactic sugar") for regular expression patterns. There is no shortage of tools that aid in the construction of such syntactic dialects (e.g. Camlp4 [13], SugarJ [5], SugarHaskell [7] and many others).

Somewhat more rarely, dialects are motivated by the need to extend a general-purpose language with new derived semantic constructs, i.e. constructs that cannot be expressed as purely syntactic sugar, but where the general-purpose language is a suitable translation target. As a minimal example, consider System **F**, i.e. the polymorphic lambda calculus (perhaps the simplest general-purpose language) [12]. If we wanted to express record types in **F**, we could not do so simply with derived syntax. We could, however, develop a dialect that included the type and term operators relevant to records, specifying their static semantics directly but their dynamic semantics by translation to the polymorphic lambda calculus (using a standard Church-style encoding).

couple more sentences

The problem with designing new language dialects in these situations is that a program written in one dialect cannot safely and idiomatically interface with libraries written in a different dialect. To do so would require combining the two dialects into a single language, but there is no systematic method to do so, much less one that guarantees that important

metatheoretic properties established about each dialect in isolation will be conserved. As a simple example, two dialects might be known to have an unambiguous concrete syntax in isolation, but if the syntax is naïvely combined, ambiguities could easily arise. Without strong modular reasoning principles to rely upon, it is difficult to construct large software systems and ecosystems. The only reasonable approach for software engineers who must program "in the large" is to eschew constructs housed in dialects and settle on a single common language, where they can rely on the guarantees provided by its type and module system. This has left many useful constructs for programming "in the small" obscure.

2 Contributions

Our aim in this thesis is to make dialect formation less frequently necessary by designing language mechanisms that give library providers the ability to express derived syntactic and semantic constructs from within the language, in a manner that ensures that these constructs can be reasoned about separately and composed arbitrarily.

To provide a coherent vehicle for these mechanisms, we will introduce a new general-purpose language called Wyvern. Let us briefly summarize its organization. Syntactically, Wyvern has a layout-sensitive textual concrete syntax (i.e. indentation is meaningful). This choice is not fundamental to our proposed mechanisms, but it will be useful for a class of examples that we will discuss later. Semantically, Wyvern consists of 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 [11]. The core language (i.e. the language of types and expressions) is split into an *external language* (EL) and an *internal language* (IL). The semantics of EL constructs are specified by a type-directed translation to the much simpler IL. Notionally, this can be thought of as simply a shifting of the first stage of a type-directed compiler into the language specification [20]. The internal language we will use is the polymorphic lambda calculus with binary product and sum types and recursive types (features like state and exceptions could also be included, but for simplicity, we will omit them) [12].

We will begin by describing Wyvern's mechanisms for expressing new forms of derived concrete syntax, assuming for the moment that the EL is otherwise specified in a standard way, in Section 3. We then turn in Section **??** to a mechanism for introducing new semantic constructs – in particular, new base types and operators – into the EL. Constructs that are built in to languages like ML, like n-ary tuples/records, labeled sums and others, can in Wyvern be realized as modes of use of this mechanism. In both cases, the mechanisms can be broadly understood as novel forms of *static metaprogramming*. Users define each new syntactic and semantic construct by writing code that generates terms and types. This code is evaluated statically, in the course of typechecking EL terms.

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 Wyvern 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 syntactic and semantic constructs can be reasoned about separately and then used together in any combination. These justify our classification of Wyvern as a *modularly extensible programming language*.

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, Wyvern 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 [21]. 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 [12]. Wyvern 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 [17]. 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" [19].

   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 [16], motivating research on more direct syntax extension mechanisms [3].

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 [15]. A theoretical account of such mechanisms has been developed by Griffin [10]. 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* [6], Camlp4 [13] 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 [18]. 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` satisying `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 Wyvern 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, Wyvern 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 Wyvern expression.

3.3.2 Type-Specific Languages (TSLs)

3.4 Timeline

SAC, ECOOP.

TODO: module-parameterized TSMs

4 Extensible Semantics

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 [9]. 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 [8]). 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 [12]. 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 [4] and records with functional update operators (also called *extensible records*) [13]. 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)
```

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] 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] 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.

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

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

[7] S. Erdweg, F. Rieger, T. Rendel, and K. Ostermann. Layout-sensitive language extensibility with SugarHaskell. In *Proceedings of the 2012 Symposium on Haskell*, pages 149–160. ACM, 2012.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

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

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

[17] 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.

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

[19] 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.

[20] 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.

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