Type-Directed, Whitespace-Delimited Parsing for Embedded DSLs

Cyrus Omar, Benjamin Chung, Darya Kurilova, Alex Potanin[1] and Jonathan Aldrich
Carnegie Mellon University
{comar, bwchung, darya, aldrich}@cs.cmu.edu, and alex@ecs.vuw.ac.nz[1]

ABSTRACT

Domain-specific languages improve ease-of-use, expressiveness and verifiability, but defining and using different DSLs within a single application remains difficult. We introduce an approach for embedded DSLs where 1) whitespace delimits DSL-governed blocks, and 2) the parsing and type checking phases occur in tandem so that the expected type of the block determines which domain-specific parser governs that block. We argue that this approach occupies a sweet spot, providing high expressiveness and ease-of-use while maintaining safe composability. We introduce the design, provide examples and describe an ongoing implementation of this strategy in the Wyvern programming language. We also discuss how a more conventional keyword-directed strategy for parsing of DSLs can arise as a special case of this type-directed strategy.

1. INTRODUCTION

Domain-specific languages (DSLs) [8] allow developers to work with specialized abstractions in a natural manner, and allow for specialized verification and compilation strategies that can improve verifiability and performance. However, for DSLs to reach their full potential, it must be simple to define a new DSL, invoke it when needed, and to use multiple DSLs within a host general-purpose language (GPL), such that pieces of DSL code can interoperate to form a complete application. These intuitions are captured by the following core design criteria that govern our work:

- *Composability*: It should be possible to use multiple DSLs and a GPL within a single program unit. Within the file-based paradigm used by most contemporary languages, this means including multiple DSLs within a single file. Moreover, it should be possible to embed code written in one DSL within another DSL when appropriate, without requiring them to have specific knowledge of each other. This should be possible without interference between DSLs used in any combination: DSLs should be *safely composable*.
- *Interoperability*: It should be possible to pass around and operate on values that were defined in foreign DSLs in a reasonably natural manner (that is, without requiring large amounts of "glue code"). Additional requirements, such as the ability to do so with the safety guarantees provided in the foreign DSL, may also be relevant in many settings.

In addition to these fundamental criteria, we believe that to be most useful, a system supporting DSLs should satisfy the following related design criteria:

- *Flexibility*: Support a variety of notations and new language mechanisms, with minimal bias;
- *Modularity*: Support defining DSLs as combinations of reusable components distributed directly within libraries;
- *Identifiability*: Make it easy for programmers to identify which code is written in which DSL and what it means;
- *Simplicity*: Keep the complexity and cost of both defining and invoking a DSL as low as possible.

We are developing a comprehensive language design, *Wyvern*, that we believe can satisfy these design criteria well, and that specifically considers language-internal extensibility from the start. In Wyvern, DSL developers define the run-time semantics of DSL constructs via translation into a common host language, as in many other DSL frameworks. The novelty of the proposed extensibility mechanism lies in the ways in which we delimit and determine the *scope* of DSL code:

- Wyvern is a *whitespace-delimited* language. Source code that is governed by a DSL, rather than the GPL, occurs in whitespace-delimited blocks and must be indented further than the GPL line introducing it. A decrease in indentation relative to the baseline of the DSL block signals its end. This scheme delimits the scope of each DSL in a clear manner, both to the programmer and the top-level parser, supporting the principle of identifiability. It also allows Wyvern to avoid restrictions on a DSL's use of delimiters internally. Because the GPL grammar is not extended in a global manner, it also guarantees that syntactic conflicts cannot arise at link-time.
- Within this basic syntactic framework, we then propose a novel type-directed dispatch mechanism: the *expected type* of an expression, rather than an explicit keyword, determines which DSL grammar should parse the delimited block that generates that expression. That is, *grammars are associated with types*. We will show that the more common keyword-directed strategy arises as a special case of this strategy.

This mechanism allows us to satisfy many of the criteria above, including safe composability, while still being quite expressive, as we will show with examples in the next section. We will continue by describing our approach in more detail (§3), discuss ongoing research directions (§4), and conclude with related work (§5).

GlobalDSL '13, Montpellier, France

```
1  val dashboardArchitecture : Architecture = ~
2     external component twitter : Feed
3        location www.twitter.com
4     external component client : Browser
5        connects to servlet
6     component servlet : DashServlet
7        connects to productDB, twitter
8        location intranet.nameless.com
9     component productDB : Database
10       location db.nameless.com
11    policy mainPolicy = ~
12       must salt servlet.login.password
13       connect * -> servlet with HTTPS
14       connect servlet -> productDB with TLS
```

Figure 1: Wyvern DSL: Architecture Specification

```
1  val newProds = productDB.query(~)
2     select twHandle
3     where introduced - today < 3 months
4  val prodTwt = new Feed(newProds)
5  return prodTwt.query(~)
6     select *
7     group by followed
8     where count > 1000
```

Figure 2: Wyvern DSL: Queries

2. MOTIVATING EXAMPLES

We start with a few examples to illustrate the expressiveness of our approach and the breadth of DSLs we plan for it to support. The examples are presented in the proposed syntax for Wyvern, a new language being developed by our group that is targeted toward building secure web and mobile applications. We will informally describe each of these examples here, and further explain how such code is parsed in Section 3.

The first example, shown in Figure 1, describes the overall architecture of a "hot product dashboard" application. The variable `dashboardArchitecture` is explicitly ascribed type `Architecture`. Rather than explicitly providing a value of this type, we instead use a DSL that makes specifying the component architecture of the application more concise and readable. This DSL code appears in the subsequent whitespace-delimited block and is introduced by a tilde (~). The example architecture declares several components, some of which are declared **`external`** to indicate that they are used by this application but are not part of it directly. Component types are declared after a colon and attributes like connectivity location, are declared after the type (formatted in an indented block for readability). The **`policy`** keyword (line 11) introduces a security policy, which constrains the communication protocols that can be used and enforces the secure handling of passwords. A separate type, `Policy`, is associated with such policies. Although we could instantiate this type explicitly using a Wyvern expression, we use a DSL for defining policies instead, again within a whitespace-delimited block introduced by a tilde.

Figure 2 shows how a DSL for database queries can be used from within ordinary Wyvern code. The example shows code for computing a feed that is derived from tweets about a company's new products. In this example, the use of a querying DSL is triggered by the use of methods named `query` expecting an argument of type `DBQuery` (line 1) or `FeedQuery` (line 5) respectively. These types define related but distinct syntax for queries, determined by the expected type of expression where the tilde appears (tildes need not appear only at the ends of lines). Queries are again delimited by indentation. This mechanism is similar to what can be expressed in languages with built-in query syntax like LINQ [2], but in this case, it is entirely user-defined, rather than built into the language.

```
1  serve(page, loc) where
2    val page = ~
3      html:
4        head:
5          title: Hot Products
6          style: {myStylesheet}
7          body:
8            div id="search":
9              {SearchBox("products")}
10           div id="products":
11             {FeedBox(servlet.hotProds())}
12   val loc = ~
13     products.nameless.com
```

Figure 3: Wyvern DSLs: Presentation and URLs

Finally, Figure 3 shows a DSL for presenting the hot product application to a web browser, served at a particular URL. Here, two DSLs are used within a single function call. To allow this without introducing ambiguity, the user can use a **`where`** clause, similar to that found in Haskell [10]. The presentation DSL is based on HTML and associated with a type, `HTMLElement`. It uses an indentation-sensitive syntax and allows integration of Wyvern code of the appropriate type using curly braces. The second DSL simply canonicalizes URL literals into Wyvern values of type `URL`.

3. APPROACH

The examples above demonstrate the core mechanism used in Wyvern: each expression or declaration can contain at most one tilde (~). The line following the term containing the tilde must begin an indented block. This block will be parsed according to a grammar associated with the expected type of the expression where the tilde occurred. In Figure 1, this type was determined by an explicit type annotation. In Figures 2 and 3, it was determined implicitly by the argument types of the function being called. Although a single tilde per expression may initially seem limiting, we can see in Figure 3 that the use of a **`where`** clause allows for the use of multiple DSLs within a single expression.

Figure 4 shows how users equip types with domain-specific grammars, here for the Architecture example in Figure 1. The grammar associated with `Architecture` is defined at the top level by a specially-named production, **`grammar`**, and it also defines two sub-productions for component and policy specifications. Each component specification includes a name, a type and an optional list of attributes. The name is specified using the **`ID`** production, which is a globally-available production that matches valid Wyvern (i.e. the GPL's) identifiers. Similarly, the **`TYPE`** production matches Wyvern types. These productions cannot be extended directly, but can be used within DSLs as needed and thus conflicts are detected when the library is compiled, rather than deferred to link-time.

One component attribute in the Architecture DSL is **`location`**. On line 4.6, we see the use of a production defined in another type, `URL`. This means that only URL literals are valid at that position, but *not* any Wyvern expression of type `URL`. To instead ask for any Wyvern expression of a particular type, we use the notation used in the **`policy`** production on line 4.8. Here, a named policy is defined as an identifier followed by an equals sign and a Wyvern expression of type `Policy`. This is denoted by the form **`EXP : T`**, where `T` is a type in scope of the grammar definition. Unlike in the URL

```
type Architecture
  grammar ::= (component|policy)+
  component ::= "external"? "component"
                ID ":" TYPE
               ((componentAttr)*)?
  componentAttr ::= "location " URL.grammar
                  | "connects to" (ID ",")* ID
  policy ::= "policy" ID "=" (EXP : Policy)
```

Figure 4: Type-Associated Grammar

example above, if there is a DSL associated with this type, it can only be used within a whitespace-delimited block introduced by a tilde. This key distinction can be seen in Figure 1. Again, it can be modularly verified that this grammar does not contain conflicts, assuming that the core Wyvern grammar does not change. We expect it to become stable relatively early in its development, with most new features introduced via the embedded DSL mechanism.

Parsing and Typechecking.

Wyvern source code is parsed in two phases. The first phase uses a standard declarative whitespace-delimited parser (e.g. [16]) where all whitespace-delimited blocks are left as unparsed "DSL literals". The second phase occurs in tandem with typechecking. When a tilde is encountered, the compiler determines the expected type where the tilde appeared (based on function signatures, or explicit type annotations) and parses the subsequent whitespace-delimited block according to the grammar associated with that type. The baseline indentation is stripped from this block, so it appears to the parser as if the DSL begins on the leftmost column of the block. Any Wyvern expressions that occur internally to a DSL (such as the policy specification in Figure 1 or curly-brace-delimited expressions in Figure 3) are also parsed and typechecked at this time. After parsing a DSL block, it must be verified and translated to Wyvern code. The mechanism for doing this is similar, at a high level, to that for defining the grammar itself, but the details of these subsequent steps are beyond the scope of this paper.

Procedural Parsing.

The grammar definition in Figure 4 is declarative and relies on a parser generator included as part of Wyvern. It may be desirable in some cases, such as when a grammar cannot easily be expressed within our declarative framework, or when an existing parser can be called into, to allow a parsing algorithms to be encoded directly. An important use case is for interoperability layers between Wyvern and other full-scale programming languages, particularly ones for which parsing is known to be difficult (e.g. C, Haskell, Python).

It can be observed that a declarative grammar inside a class definition can be seen as inducing a class method (that is, an operation defined on the class itself, which can be invoked by the compiler or run-time system – a concept borrowed from Smalltalk [9]) called `parse`, that transforms a string to some AST representation. This lower-level interface is exposed directly to programmers who wish to specify parsing in a procedural manner.

4. DISCUSSION AND FUTURE WORK

Keyword-Directed Invocation.

In most DSL frameworks, a switch to a DSL is indicated by a keyword or function call naming the DSL to be used. Wyvern eliminates this overhead in many cases by determining the DSL based on the expected type of an expression. This lightweight mechanism is particularly useful for small DSLs, like the one associated with `URL`. Keyword-directed invocation of a DSL is simply a special case of this approach. In particular, a keyword macro can be defined as a function with a single argument of a type specific to that keyword. The type contains the implementation of the domain-specific syntax associated with that keyword. In the most general sense, it may simply allow the entire **`EXP`** grammar, manipulating it in later phases of compilation.

As an example, consider control flow operators like `if`. This can be defined as a polymorphic method of the `bool` type with signature $(\texttt{unit} \to \alpha, \texttt{unit} \to \alpha) \to \alpha$. That is, it takes the two branches as functions and chooses which to invoke based on the value of the boolean, using perhaps a more primitive control flow operator, like case analysis, or even a Church encoding of booleans as functions. In Wyvern, the branches could be packaged together into a type, `IfBranches`, with an associated grammar that accepts the two branches as unwrapped expressions. Thus, `if` could be defined entirely in a library and used as follows:

```
<guard>.if(~)
  then
    <any EXP>
  else
    <any EXP>
```

For methods like `if` where constructing the argument explicitly will almost never be done, it may be useful to mark the method in a way that allows Wyvern to assume it is being called with a DSL argument immediately following its use. This would eliminate the need for the `(~)` portion, supporting even more conventional notation. We have not considered this possibility in detail.

Explicit Delimiters.

Throughout this paper, DSLs have been delimited by whitespace. This allows arbitrary syntax within DSLs, since no delimiters need to be reserved to indicate the end of the DSL and thus there is no need for escaping internal uses of these delimiters. In cases where DSL expressions are expected to be reasonably short, such as the `URL` example, or where delimiters are more natural than whitespace, such as for array or dictionary literals, it may be desirable to support other forms of delimited "DSL literals".

One possible strategy for this is to reserve a number of common delimiter forms, such as quotation marks and forms of braces, as equivalent DSL literal forms. The traditional meaning of these delimiters, such as quotation marks for strings and square brackets for lists, would then simply be convention in Wyvern. That is, the following expressions, as well as several similar ones, would be precisely equivalent (the programmer could choose the most convenient form, given the enclosed term):

```
f("http://github.com/wyvernlang")
f([http://github.com/wyvernlang])
```

Alternatively, types could specify the set of permitted delimiters so that conventions can be enforced by the compiler, improving identifiability. We have not yet explored either of these possibilities in detail, nor explored options that allow *arbitrary* type-specified delimiters (a naive strategy for which would require that the first phase of parsing also be type-directed, which we wish to avoid).

Interaction with Subtyping.

The mechanism described here does not consider the case where multiple subtypes of a base type define a grammar. This can be resolved in several ways. We could require that only the *declared* type's grammar is used (if a subtype's grammar is desired, an explicit type annotation on the tilde can be used). Alternatively, we

could attempt to parse against all relevant subtypes, only requiring explicit disambiguation when ambiguities arise. Wyvern does not currently support subtyping, so we leave this as future work.

5. RELATED WORK

The most well-known mechanism for extending languages is macros, as exemplified by hygienic macros in Scheme. Macros in Scheme and other Lisp-style languages are written in the language itself and benefit from its simple syntax – parentheses universally serve as expression delimiters (although proposals for whitespace as a substitute for parentheses have been made [13]). Our work is inspired by this flexibility, but aims to support richer syntax as well as static types. Wyvern's use of types to trigger parsing avoids the overhead of needing to invoke macros explicitly by name and makes it easier to compose DSLs declaratively.

Some language extensibility projects provide metaprogramming facilities at levels of abstraction above parsing. For instance, OJ (previously, OpenJava) [18] provides a macro system based on a meta-object protocol, and Backstage Java [15], Template Haskell [17] and others employ compile-time meta-programming. Each of these systems provide macro-style rewriting of source code, but they provide at most limited extension of language parsing.

Other systems aim at providing forms of syntax extension that change the base language, as opposed to our whitespace-delimited approach. For example, Camlp4 [6] is a preprocessor for OCaml that offers the developer the ability to extend the concrete syntax of the language via the use of parsers and extensible grammars. SugarJ [7] takes a library-centric approach which supports syntactic extension of the Java language by adding libraries. In Wyvern, the core language (particularly the **`EXP`** sort) is not extended directly, so conflicts cannot arise at link-time.

Scoping DSLs to expressions of a single type comes at the expense of some flexibility, but we believe that many uses of DSLs are of this form already. A previous approach has considered type-based disambiguation of parse forests for supporting quotation and anti-quotation of arbitrary object languages [3]. Our work is similar in spirit, but does not rely on generation of parse forests and associates grammars with types, rather than types with grammar productions. We believe that this is a more simple and flexible methodology. The remaining approaches to syntax extension, such as XJ [4] are keyword-directed in some form. We believe that a type-directed approach is more seamless and general, sacrificing a small amount of identifiability in some cases.

Researchers have also developed DSL frameworks and language workbenches, including MPS [1], Spoofax [11], Ensō [5] and others [12, 19] that provide support for generating new programming languages and tooling in a modular manner. Compared to these approaches, Wyvern focuses on extensibility *internal* to the language, rather than taking an approach where each DSL is *external* relative to the host language, improving interoperability and composability.

Finally, recent work on Active Code Completion is related to this work in that it associates code completion palettes with types [14]. Such palettes could be used for defining a DSL syntax for types. However, that syntax is immediately translated to Java syntax at edit-time while this work integrates with the core parsing facilities of the language.

Acknowledgements

We thank the anonymous reviewers for helpful comments, and acknowledge the support of the Department of Defense and the Air Force Research Laboratory. CO is supported by the NSF Graduate Research Fellowship.

6. REFERENCES

[1] JetBrains MPS – Meta Programming System. `http://www.jetbrains.com/mps/`.

[2] LINQ (Language-Integrated Query). `http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx`.

[3] M. Bravenboer, R. Vermaas, J. Vinju, and E. Visser. Generalized type-based disambiguation of meta programs with concrete object syntax. In *Generative Programming and Component Engineering*, pages 157–172. Springer, 2005.

[4] T. Clark, P. Sammut, and J. S. Willans. Beyond annotations: A proposal for extensible java (xj). In *SCAM*, pages 229–238, 2008.

[5] W. R. Cook, A. Loh, and T. van der Storm. Ensō: A self-describing DSL workbench. `http://enso-lang.org/`.

[6] D. de Rauglaudre. *Camlp4 - Reference Manual*, Sep 2003.

[7] S. Erdweg, L. C. Kats, T. Rendel, C. Kästner, K. Ostermann, and E. Visser. Sugarj: library-based language extensibility. In *OOPSLA*, pages 187–188. ACM, 2011.

[8] M. Fowler and R. Parsons. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[9] H. H. P. III. Smalltalk: A White Paper Overview. March 2003.

[10] S. L. P. Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[11] L. C. L. Kats and E. Visser. The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. In *OOPSLA*, pages 444–463. ACM, 2010.

[12] H. Krahn, B. Rumpe, and S. Völkel. Monticore: Modular development of textual domain specific languages. In *Objects, Components, Models and Patterns*, pages 297–315. Springer, 2008.

[13] E. Möller. SRFI-49: Indentation-sensitive syntax. `http://srfi.schemers.org/srfi-49/srfi-49.html`, 2005.

[14] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 859–869. IEEE Press, 2012.

[15] Z. Palmer and S. F. Smith. Backstage Java: Making a Difference in Metaprogramming. In *OOPSLA '11*, pages 939–958, New York, NY, USA, 2011. ACM.

[16] R. L. O. Rule and M. D. Adams. Principled parsing for indentation-sensitive languages. 2013.

[17] T. Sheard and S. P. Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.

[18] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A Class-based Macro System for Java. In *Reflection and Software Engineering*, pages 117–133. Springer, 2000.

[19] M. G. J. van den Brand. *Pregmatic: A Generator for Incremental Programming Environments*. PhD thesis, Katholieke Universiteit Nijmegen, 1992.