Naturally Embedded DSLs

Jonathan Aldrich
Carnegie Mellon University
aldrich@cs.cmu.edu

Alex Potanin
Victoria University of Wellington
Alex.Potanin@ecs.vuw.ac.nz

Abstract

Domain-specific languages can be embedded in a variety of ways within a host language. The choice of embedding approach entails significant tradeoffs in the usability of the embedded DSL. We argue embedding DSLs *naturally* within the host language results in the best experience for end users of the DSL. A *naturally embedded DSL* is one that uses natural syntax, static semantics, and dynamic semantics for the DSL, all of which may differ from the host language. Furthermore, it must be possible to use DSLs together naturally—meaning that different DSLs cannot conflict, and the programmer can easily tell which code is written in which language.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords naturally embedded domain specific languages

1. Usability Challenges in Embedded DSLs

Domain-specific languages (DSLs) can provide a number of advantages in software development. Developers can express a program in the most natural way for the domain they are working in. Conversely, the DSL compiler can take advantage of domain knowledge to catch programming errors and produce more optimized code, compared to a general-purpose language and compiler.

Domain-specific languages often describe part of a solution to a larger problem, and solving the larger problem may therefore require more than one DSL. For example, a program may solve a problem that involves two different domains and therefore two different DSLs to capture them, or else a single domain may have different aspects that are best expressed with different DSL notations. In either of these cases, it is important to semantically integrate the DSLs with each other, and potentially with a general-purpose language as well.

A natural approach to integration is to *embed* DSLs within a single general-purpose host language. Embedded DSLs can be defined in multiple ways. In a library-based embedding approach, the DSL is defined as a library in the host language, so that expressions using that library make up a kind of DSL. In a macro-based embedding approach, macros are defined for DSL constructs, and the compiler expands uses of the macros into host language expressions, which are then typechecked and executed as usual.

[Copyright notice will appear here once 'preprint' option is removed.]

Both of these approaches have usability problems. The host language or its macro system may not support the most appropriate syntax for the DSL, in which case DSL programs must be expressed awkwardly. If the programmer makes a semantic mistake in the DSL, the mistake will be caught after translation into the host language (if it is caught at all), meaning that the error message may be difficult for the programmer to understand.

A third approach is extensible languages, in which a language can be extended through a library[1] with new syntax and semantics [2]. The extensible language approach can address both of the problems above, but can introduce new usability problems. Depending on the design of the extensible language, it may be difficult to know what code is being expressed in the host language vs. in an extension. Furthermore, different DSLs may conflict when the programmer tries to use them together, forcing the programmer to disambiguate the uses somehow.

Note that we focus on usability for programmers who use the host language and its embedded DSLs. Making it easy to define embedded DSLs is a different usability problem. Solving both problems are worthwhile, but we prioritize the usability for programmers who use DSLs because a DSL is used many more times than it is defined, and because programmers who define DSLs are likely to have more expertise (and thus more ability to cope with complexity) than programmers who use them.

2. Naturally Embedded DSLs

We argue that DSLs should be embedded *naturally* within the host language. A system supporting *naturally embedded DSLs* has the following characteristics:

- The ability to define **new syntax** that expresses the ideas in a domain in the most natural way. The host language should place as few as possible restrictions on the syntax of embedded DSLs.
- The ability to define **new semantics** that can be understood in terms of the target domain, rather than as a mode of use of the host language. Note that the DSL may still be implemented by translation into the host language, or by an interpreter written in the host language; what is important is that this translation or interpretation is not visible to the end user.
- The ability to define **error messages** (both at compile time and at run time), **introspection** and **debugging** in terms of the target domain, rather than in terms of the host language.
- Support for the programmer to easily **identify** what language each expression is written in, and what host-language type each DSL expression has.
- A **modularity guarantee** that separately-defined DSLs can be used together without conflicts.

[1] or a compiler plug-in, although this makes the DSL dependent on the particular compiler used rather than on a language standard that can be implemented by multiple compilers

```
def mostOptimalTrade() : ShareTransaction
  {pick best 42-shot strategy
      from {APPL, XRO, MSFT}
    apply strategy for next 10 day range
    only keep the indexed shares}

def storedTrade(t : ShareTransaction) : SQLQuery
    {CREATE in "TRADES" WHERE (t.name, t.buy, t.sell)}
```

Figure 1: A simple example mixing a share trading and database DSL's in Wyvern.

3. Extended example: Wyvern

As part of our talk, we will describe the Wyvern programming language as an example of the naturally embedded domain-specific language approach [5] that was inspired by earlier work by Omar et al. [4] Wyvern is a modularly extensible language in which language extensions can be defined as libraries.

Wyvern allows DSLs to be defined with their own **syntax**, can be an arbitrary sequence of characters that is distinguished from the host language using indentation. Delimiters such as { and } can also be used, at the cost of requiring the DSL to balance any occurrences of these within its syntax. Extensions define their own **semantics**, including custom error messages.[2] Our approach of using indentation or reserved delimiters distinguishes the DSL from the host language, and we use expected types or macro names so that developers can easily **identify** which DSL is being used in each subexpression. The delimiting and identification mechanisms also work together to ensure that DSLs **cannot conflict** with one another.

Our talk will include a brief demonstration of Wyvern that illustrates these features. Figure 1 demonstrates how a banking app can mix a specific trading language with a database manipulation language within a common language of Wyvern.

4. Conclusion

Related approaches to our work include tools such as *ProteaJ* [3] that uses *protean operators* that resolve potential syntactic conflicts by looking at expected types. An alternative approach uses keywords to distinguish different embedded DSLs (like XJ [1]).

We believe that the *natural* approach to DSL embedding, for example as in the Wyvern language, will significantly increase the usability of DSLs and thereby help software developers realize more of their benefits in practice.

References

[1] Tony Clark, Paul Sammut, and James S. Willans. Beyond annotations: A proposal for extensible Java (XJ). In *Source Code Analysis and Manipulation*, 2008.

[2] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: library-based language extensibility. In *Object-Oriented Programming Systems, Languages, and Applications*, 2011.

[3] Kazuhiro Ichikawa and Shigeru Chiba. Composable user-defined operators that can express user-defined literals. In *Modularity*, 2014.

[4] C. Omar, YoungSeok Yoon, T.D. LaToza, and B.A. Myers. Active code completion. In *Proc. 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'11)*, pages 261–262, 2011.

[5] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely composable type-specific languages. In *ECOOP*, pages 105–130. Springer-Verlag, 2014.

[2] support for customizable error messages is still being implemented as of this writing, but hopefully will be complete for a demo at DSLDI