Composable Syntax Macros

Stanley Wang, Cyrus Omar and Jonathan Aldrich
Carnegie Mellon University
{stwong,comar,aldrich}@cs.cmu.edu

1 Introduction

One way domain-specific languages can benefit users is to provide them with a clear and familiar syntax while retaining the semantics of a powerful underlying general-purpose language. However, in a complicated project, problems from different domains come up together, so no DSL alone is sufficient. Common approaches for embedding DSLs into general-purpose languages include attempting to repurpose existing syntax to make an API easy to use, or encoding DSLs as string arguments to DSL engines. But both ways have their flaws. Exporting complicated library calls always leads to complex code and a sometimes less ideal interface, while using strings to represent DSL code is less safe and can still be awkward, due to conflicts between string syntax and the domain-specific syntax. Approaches that permit library providers to directly extend the base syntax of a language can lead to ambiguities.

In our previous work on *type-specific languages (TSLs)* [2], we introduced techniques for associating arbitrary syntax extensions with types. By pushing the parsing of delimited blocks of specialized syntax into the typechecker, we guaranteed that these extensions could be composed arbitrarily. For example, Figure 1 (left) shows an example of a TSL associated with the `XML` type (TSLs can be seen as metadata associated with a type; see [2]).

However, this work was limited in two important ways: 1) only a single choice of syntax was available for any particular type; and 2) this couldn't be used to implement operations *on* existing values of a type, only operations that introduced values of a type. Here, we extend our previous work with *syntax macros* - explicit user-defined keywords that determine when to switch between the host languages and a DSL. By using the same strategies for delimiting specialized syntax, and the same mechanisms for ensuring hygiene, we will maintain the guarantee that syntax extensions can be composed while substantially increasing the expressiveness of our language (Wyvern). By borrowing a technique from Scala's macro system (which doesn't permit new syntax), we will retain the ability to reason about the type of a term without performing expansion in many cases.

2 Syntax Macros in Wyvern

In this section, we show several examples built in Wyvern to illustrate the usage of both *black-box keywords* and *white-box keywords* (this terminology is taken from Scala's macro system [1]). A black-box keyword is declared with a return type, so the return type is determined without referring to the expression it elaborates to. A white-box keyword does not have this restriction as the return type of the white-box keyword will be determined by the internal expression after literal transformation.

We begin in Figure 1 (right) with an example of a black-box keyword that constructs a value of type `XML` using a new syntax other than the one associated directly with a type, here simplifying XML by using whitespace to close tags rather than explicit closing tags. The tilde (~) serves a role analagous to its role for TSLs: it indicates that the body of the syntax macro is

the layout-delimited block beginning on the next line. The keyword `simpleXML` is declared on lines 1-3 with the return type (`XML`) and a definition of the parser itself (omitted).

```
1  type XML = casetype
2    ...
3    metadata = new : HasTSL
4      val parser = ~
5        ... (parser for standard XML syntax)
6  val x : XML = ~
7    <book id=1>
8      <title>XXX</title>
9      <author>XXX</author>
10   </book>
```

```
1  keyword simpleXML : XML = new
2    val parser = ~
3      ... (parser for simple XML syntax)
4  val x : XML = simpleXML ~
5    >book[id=1]
6      >title XXX
7      >author XXX
```

Figure 1: Building a value of type XML using a TSL providing the standard syntax (left) or using a "black-box" expression keyword providing a simpler layout-sensitive syntax (right).

```
1  type Bool = casetype
2    True
3    False
4    keyword if = new
5      val parser = fn self => ~
6        EXP BOUNDARY 'else' BOUNDARY EXP
7          fn e1, e2 => ~
8            case %self%
9              True => %e1%
10             False => %e2%
11 x:Bool = ...
12 x.if {branch_1} else {branch_2}
```

Figure 2: "White-box" keyword `if` defined as a member of type `Bool`

Another example presented in Figure 2 shows a white-box keyword, `if`. The type `Bool` is defined as a casetype with two cases, namely `True` and `False`. The general way to use a value of type `Bool` is by case analysis, explicitly naming these two constructors. We define the more conventional conditional operation `if` in terms of case analysis, here as a "method macro" associated with `Bool`, seen being used on line 12. Here, we are using an extension to the concrete syntax presented in [2] that supports "multi-part" bodies. Any delimited forms, e.g. `{branch_1}`, appearing next to each other, or with keywords like `else` between them, will be merged into a single body with a special `BOUNDARY` token inserted between them. The grammar, on line 6, can recognize these boundaries to separate out the parts of the keyword. This allows for a Smalltalk-style syntax for invoking macros like these. The same syntax can be used to construct multi-part TSL literals. Details of these and other examples, as well as a type-theoretic formalization of this mechanism based on that presented in [2], will be presented in our talk.

References

[1] E. Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, New York, NY, USA, 2013. ACM.

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