Objectifying a Metaprogramming Language

Vadim Zaytsev
Raincode Labs, Brussels, Belgium
vadim@grammarware.net

ACM Reference format:
Vadim Zaytsev. 2017. Objectifying a Metaprogramming Language. In *Proceedings of the -2th Workshop on New Object-Oriented Languages, Vancouver, Canada, October 2017 (NOOL'17),* 1 pages.

Abstract

Introducing object-oriented functionality is well-studied in the context of reverse and re-engineering for at least two decades [2–4], and on the instance level can usually be automated at least to some extent by object/component identification and wrapping techniques. In this abstract we would like to consider the same problem for forward engineering and apply it on the language level.

We have implemented a language extension for Rascal metaprogramming language [1]. Rascal combines features and syntax from Java, Haskell, SDF and some other languages, as well as implements some design patterns as built-in language constructs: it is a comfortable language for prototyping compilers and similar tools for program analysis and translation. Our primary goal was to create a non-disrupting extension that would "blend in" and allow to use the classic features of Rascal naturally together with OO code. The result is called **BOOL**, short for Binary Object-Oriented Language, because each statement of the language is a binary binding. The prototype is publicly available at http://github.com/grammarware/bool.

Rascal's syntax for defining composite types seemed like a good notation, so we extended it to cover concrete syntax constructs to create a sense of symmetry. A BOOL programmer can write "`Foo := plus[word] ~ list[str]`", to automatically generate a concrete syntax for parsing layout-separated sequences of words (the left hand of the binding), an abstract syntax to represent them as lists of strings (the right hand), and constructors to create an abstract instance from any concrete instance (parse tree), or its textual (parsable) representation, or pure data. **Classes** are defined the same way, e.g. "`Point := seq[int x, comma, int y] ~ class[int x, int y, method add, method sub]`", and require a separate declaration for each of the **methods**, which are bindings from function signatures to code, e.g., "`Point.add := fun[Point l, Point r] ~ Point[x:=l.x+r.x; y:=l.y+r.y]`". Such a class gets compiled to an algebraic data type, which allows native Rascal code to access fields and methods with a dot notation, but requires adding a disposable constructor name.

To make the extension more elegant and efficient, we allow to split classes into records and clusters. **Records** contain only data, cannot be recursive and thus are easily compilable to Rascal tuples, e.g., "`Pair := seq[int x, comma, int y] ~ record[int x, int y]`". **Clusters** contain only code of what can be seen as static methods in other languages, e.g., "`Point := Pair ~ cluster[method add, method sub]`", which gets compiled to another tuple type. In this example, a Rascal programmer can then instantiate a Pair and use it as a Point, e.g. "`Point.add(newPair(1,2),newPair(3,4))`". If the left hand of the cluster binding is a record name, nothing special happens, but if it is another cluster, the new cluster **inherits** all the methods of the original one, e.g., "`Complex := Point ~ cluster[method mul]`" will assume `Complex.add` and `Complex.sub` to have the same semantics as `Point.add` and `Point.sub`, respectively, unless explicitly overridden.

The absence of the right hand of a binding is denoted by a dot and is useful for defining negligible concrete syntax, e.g., "`Layout := or[space, tab, newline] ~ .`".

The result is an extremely concise DSL that compiles BOOL bindings to Rascal grammars, data types and mappings, and allows to use OO-like abstractions and techniques in accompanying Rascal code. It remains to be seen what the boundaries are of the chosen implementation strategy, and in general how it compares to alternatives (such as tuples of data-encapsulating closures), but a presentation can include demonstration of many interesting implementation details.

References

[1] P. Klint, T. van der Storm, and J. J. Vinju. 2009. Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. In *SCAM*. IEEE, 168–177. https://doi.org/10.1109/SCAM.2009.28

[2] G. A. Di Lucca, A. R. Fasolino, P. Guerra, and S. Petruzzelli. 1997. Migrating Legacy Systems towards Object-Oriented Platforms. In *ICSM*. IEEE, 122–129. https://doi.org/10.1109/ICSM.1997.624238

[3] P. Newcomb and G. Kotik. 1995. Reengineering Procedural into Object-Oriented Systems. In *WCRE*. IEEE, 237–249. https://doi.org/10.1109/WCRE.1995.514712

[4] S. Pidaparthi and G. Cysewski. 1997. Case Study in Migration to Object-Oriented System Structure Using Design Transformation Methods. In *CSMR*. IEEE, 128–135. https://doi.org/10.1109/CSMR.1997.583021

NOOL'17, October 2017, Vancouver, Canada