Evolution of Metaprograms, or How to Transform XSLT to Rascal

Vadim Zaytsev
vadim@grammarware.net

Universiteit van Amsterdam, The Netherlands

Abstract

Metaprogramming is a well-established methodology of constructing programs that work on other other programs, analysing, parsing, transforming, compiling, evolving, mutating, transplanting them. Metaprograms themselves evolve as well, and there are times when this evolution means migrating to a different metalanguage. This fairly complicated scenario is demonstrated here by considering a concrete case of porting several rewriting systems of grammar extraction from XSLT to Rascal.

Metaprogramming is a well-established methodology of constructing programs that work on other other programs [8], analysing [1], parsing [15], transforming [2], compiling [3], visualising [7], evolving [4], composing [9], mutating [5], transplanting [10] them. Metaprograms themselves evolve as well, and there are times when this evolution means migrating to a different metalanguage. For example, a unidirectional chain of grammar/metamodel transformation steps can be turned into a bidirectional one (e.g., XBGF scripts to ΞBGF scripts [14]) — on the level of language instances this means turning a migration path (take X, transform into Y, forget X) into an executable relationship (change X, update Y, change Y, update X, ...). The general problem is too big to handle at the moment: we have recently successfully considered a much more focused problem of migration between metasyntaxes for grammars [11]; the focus in this abstract is on migrating grammar-mapping metaprograms.

SLPS [16], of Software Language Processing Suite, was a repository that served as a home for many experimental metaprograms — to be more precise, metagrammarware for grammar recovery, analysis, adaptation, visualisation, testing. Around 2012, final versions of such tools were reimplemented as components in a library called GrammarLab [13]: the code written in Haskell, Prolog, Python and other languages, was ported to Rascal [8], a software language specifically developed for the domain of metaprogramming.

Grammar extraction is a metaprogramming technique which input is a software artefact containing some kind of grammatical (structural) knowledge — an XML schema, an Ecore metamodel, a parser specification, a typed library, a piece of documentation — and recover the essence of those structural commitments, typically in a form of a formal grammar with terminals, nonterminals, labels and production rules [12]. Over the years the SLPS acquired over a dozen of such extractors, plus a couple of more error-tolerant recovery tools. Several of them were essentially mappings from various XML representations (XSD, EMF, TXL, etc), implemented — quite naturally — in XSLT [6].

A fragment of such a grammar extractor mapping is given on Figure 1(a). Readers that can overcome the overwhelming verbosity of the XML syntax, can see two templates that match elements `eLiterals` and `eStructuralFeatures` correspondingly, and generate output elements by reusing information harvested from specific places within the matched elements. As a language for metaprogramming and structured mapping in general, XSLT is pretty straightforward and provides functionality for branching, looping, traversal controls,

Copyright © by the paper's authors. Copying permitted for private and academic purposes.

In: A. H. Bagge (ed.): Proceedings of SATToSE 2015, Mons, Belgium, 6–8 July 2015, to be published at http://ceur-ws.org

etc, without going too deep into more complex metaprogramming practices such as naturally recursive rewriting systems or bottom-up traversals. It is also worth noting that XSLT is an untyped software language, so there is no explicit validation that all constructs matched and all constructs produced are type safe.

If we assume that all the types from the input as well as the output schemata are expressed as Rascal algebraic data types, and all the named templates invoked in this snippet are successfully mapped to Rascal functions, then these matched templates will be expressed as pattern-driven dispatched functions in Rascal such as the ones shown on Figure 1(b).

During the case studies on the existing XSLT-based grammar extractors, we found out that the following metaprogramming idioms are equally easy to express in XSLT and Rascal:

- matched template and `apply-templates` — a pattern-dispatched call of the general transform function
- named template and `call-template` — a call to a dedicated possibly polymorphic function
- `choose`, `when`, `otherwise`, `if` — pattern-driven dispatch or explicit matching with `:=` if the conditions are too deep
- `for-each` — list comprehensions

The following features were hard to match:

- Empty `when`s: in XSLT, one can easily loop through input elements and in some cases decide to return nothing by performing `<xsl:when test="..." />` — this is realised somewhat awkwardly in Rascal with the classic FP idiom of a "poor man's Maybe" (a list which is either empty or a singleton) and inlining.
- Library functions: luckily, early versions of XSLT are quite poor with respect to library functions. However, since XSLT is not by design a language for metaprogramming, its functions are also suboptimal for that domain — the conclusion was that finding a close match or writing a wrapper is almost always less preferable than a manual rewrite of the fragment in question.
- Variables: XSLT is a declarative language which allows fake elements that initialise named variables with certain values to be used later. Despite being multiparadigmatic, Rascal clearly distinguishes between Haskell-like straightforward function style and Java-like imperative style.
- Choices: surprisingly, the idiom `<xsl:template match="a|b">...</...>` was quite prevalent yet had absolutely no close equivalent in Rascal. Finally, the mapping of such matches was realised with an external function that was called separately for each of the matches.
- XPath: XSLT uses XPath expressions both in matches and access points; Rascal uses different notations for those two paradigms. It always strictly distinguishes between matches possibly yielding a set/list or a single element, while XPath always returns a possibly empty set of nodes which is incorporated in XSLT by implicit looping in some cases and by more unexpected workarounds in others. For example, `<xsl:apply-templates select="a"/>` is a loop, but `<xsl:value-of select="a"/>` is a concatenation — the latter is almost never the intended result.

Apart from these issues and some type inference for the `value-of`s, the mapping from XSLT to Rascal was quite possible to implement to migrate the bulk of the code and provide the opportunity to finish the job manually. The real extent of the work and the limitations of this approach in general are not yet studied in enough detail. One of the interesting remaining open questions is about the nature of the mismatches between the two metaprogramming platforms — are they intentional? Should the two learn from each other, or from the migration path itself?

```
<xsl:template match="eLiterals">
    <bgf:expression>
        <selectable>
            <selector>
                <xsl:value-of select="@name"/>
            </selector>
            <bgf:expression>
                <epsilon/>
            </bgf:expression>
        </selectable>
    </bgf:expression>
</xsl:template>
<xsl:template match="eStructuralFeatures">
    <xsl:choose>
        <xsl:when test="./@xsi:type='ecore:EReference'">
            <xsl:call-template name="mapEReference">
                <xsl:with-param name="ref" select="."/>
            </xsl:call-template>
        </xsl:when>
        <xsl:when test="./@xsi:type='ecore:EClass'">
            <xsl:call-template name="mapEClass">
                <xsl:with-param name="class" select="."/>
            </xsl:call-template>
        </xsl:when>
        <xsl:when test="./@xsi:type='ecore:EAttribute'">
            <xsl:call-template name="mapEAttribute">
                <xsl:with-param name="attr" select="."/>
            </xsl:call-template>
        </xsl:when>
        <xsl:otherwise>
            <terminal>
                <xsl:text>!!!</xsl:text>
                <xsl:value-of select="./@xsi:type"/>
            </terminal>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

(a)

```
GExpr transform(eLiterals(str name)) = label(name,epsilon());
GExpr transform(n:eStructuralFeatures("ecore:EReference")) = mapEReference(n);
GExpr transform(n:eStructuralFeatures("ecore:EClass")) = mapEClass(n);
GExpr transform(n:eStructuralFeatures("ecore:EAttribute")) = mapEAttribute(n);
default GExpr transform(n:eStructuralFeatures(str xsitype)) = terminal("!!!<xsitype>");
```

(b)

Figure 1: The same fragment of Ecore to BNF-like Grammar Format mapping in **(a)** XSLT and **(b)** Rascal. Besides the apparent shrink in size and the boost to readability linked to it, the latter fragment is strongly typed and thus can be automatically validated for its grammatical commitments to both the input and the output. Technically, the second fragment is still imperfect in the sense that it does not implement namespaces as types (just as substrings), which leaves a small door for bugs open.

References

[1] J. Bergeretti and B. Carré. Information-Flow and Data-Flow Analysis of while-Programs. *ACM ToPLaS*, 7(1):37–61, 1985.

[2] J. R. Cordy. The TXL Source Transformation Language. *SCP*, 61(3):190–210, 2006.

[3] D. Grune, K. van Reeuwijk, H. E. Bal, C. J. Jacobs, and K. G. Langendoen. *Modern Compiler Design*. Springer, 2012.

[4] T. Gîrba, J.-M. Favre, and S. Ducasse. Using Meta-Model Transformation to Model Software Evolution. *ENTCS*, 137(3):57–64, 2005.

[5] Y. Jia and M. Harman. Higher Order Mutation Testing. *Information & Software Technology*, 51(10):1379–1393, 2009.

[6] M. Kay. XSL Transformations (XSLT) Version 2.0. *W3C Recommendation*, 23 January 2007. www.w3.org/TR/2007/REC-xslt20-20070123.

[7] P. Klint, B. Lisser, and A. van der Ploeg. Towards a One-Stop-Shop for Analysis, Transformation and Visualization of Software. In A. M. Sloane and U. Aßmann, editors, *SLE*, volume 6940 of *LNCS*, pages 1–18. Springer, 2012.

[8] P. Klint, T. van der Storm, and J. J. Vinju. EASY Meta-programming with Rascal. In J. M. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, editors, *Revised Papers of the Third International Summer School on Generative and Transformational Techniques in Software Engineering*, volume 6491 of *Lecture Notes in Computer Science*, pages 222–289. Springer, 2009.

[9] P. Klint, J. Vinju, and T. van der Storm. Language Design for Meta-programming in the Software Composition Domain. In A. Bergel and J. Fabry, editors, *Software Composition*, volume 5634 of *LNCS*, pages 1–4. Springer, 2009.

[10] A. Marginean, E. Barr, J. Petke, Y. Jia, and M. Harman. Automated Software Transplantation. In *ISSTA*, 2015. In print.

[11] V. Zaytsev. Language Evolution, Metasyntactically. *EC-EASST; BX*, 49, 2012.

[12] V. Zaytsev. Notation-Parametric Grammar Recovery. In A. Sloane and S. Andova, editors, *Post-proceedings of LDTA 2012*. ACM DL, June 2012.

[13] V. Zaytsev. GrammarLab: Foundations for a Grammar Laboratory, 2013–2015. http://grammarware.github.io/lab.

[14] V. Zaytsev. Case Studies in Bidirectionalisation. In *Pre-proceedings of TFP*, pages 51–58, May 2014. Extended Abstract.

[15] V. Zaytsev and A. H. Bagge. Parsing in a Broad Sense. In *Proceedings of MoDELS*, volume 8767 of *LNCS*, pages 50–67. Springer, Oct. 2014.

[16] V. Zaytsev, R. Lämmel, T. van der Storm, L. Renggli, R. Hahn, and G. Wachsmuth. Software Language Processing Suite[1], 2008–2014. http://slps.github.io.

[1]The authors are given according to the list of contributors at http://github.com/grammarware/slps/graphs/contributors.