Evolution of Metaprograms: XSLT as a Metaprogramming Language

Vadim Zaytsev, vadim@grammarware.net

Universiteit van Amsterdam, The Netherlands
Raincode, Belgium

Abstract

Metaprogramming is a methodology of constructing programs that analyse and transform other programs. Metaprograms themselves evolve as well, and there are times when this evolution means migrating to a different metalanguage. We illustrate this complicated scenario by a concrete case of porting grammar manipulation scripts from XSLT to Rascal, and list common metaprogramming features from XSLT used in the corpus.

1 Problem and Context

Metaprogramming is a well-established methodology of constructing programs that work on other programs [20], analysing [4], parsing [38], transforming [8], compiling [15], visualising [19], evolving [13], composing [22], mutating [17], transplanting [3] them.

A typical metaprogram operates on some kind of structural representation of another program, which it constructs (by parsing) or takes by interoperating with the previous tool in the chain. It is traditional for such representations to be close to trees because termination proofs on tree traversals are much easier than on graph rewritings. In some cases the term metaprogramming refers to the program transforming itself (with reflection, staging, morphing, etc), in which case it is a form of generative programming [9]. Parsers, compilers, XML validators, code linters, refactoring tools, software analysis and visualisation plugins, IDEs are all examples of metaprograms.

In: A.H. Bagge, T. Mens (eds.): Postproceedings of SATToSE 2015 Seminar on Advanced Techniques and Tools for Software Evolution, University of Mons, Belgium, 6-8 July 2015, published at http://ceur-ws.org

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 [37]) — 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 [34]; the focus in this project was on migrating grammar-mapping metaprograms.

SLPS [39], or 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 [36]: the code written in Haskell, Prolog, Python and other languages, was ported to Rascal [20], a software language specifically developed for the domain of metaprogramming.

Grammar extraction [35] is a metaprogramming technique which consumes 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 recovers the essence of those structural commitments, typically in a form of a formal grammar with terminals, nonterminals, labels and production rules. Over the years the SLPS acquired over a dozen of such extractors, plus a couple of more error-tolerant recovery tools. The grammar extractors of SLPS were intentionally implemented in a *subjective* way, using the means from the *same* technological space where grammars originated from: ANTLR was used to parse ANTLR grammars, TXL parsed TXL, MetaEnvironment was handling SDF specifications,

etc. Several of them were essentially mappings from various XML representations (XSD, EMF, TXL, etc), implemented, quite naturally, in XSLT [18] to be used with `xsltproc` [12] working on top of `libxslt` [31].

This paper is a preliminary report on reverse engineering metaprogramming features used in **43** XSLT scripts of SLPS, used for transformation, mapping and pretty-printing of various trees. To complete it, **10568** lines of those scripts were manually inspected, which led to identification of **74** common patterns. Then the scripts were manually annotated by tags referring to those patterns, **4978** annotations in total. Insights into the ways XSLT is used in practice for metaprogramming activities, can be leveraged to find missing features in proper language workbenches [11], since we claim the application domain be the same (i.e., metaprogramming).

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, etc., without going too deep into more complex metaprogramming practices such as naturally recursive rewriting systems or advanced traversal strategies. 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.

GrammarLab grammar extractors are *objective*: they are all implemented in *one* metaprogramming language (Rascal [20, 21]). This is a paradigm different from the one adopted by SLPS that we explained earlier, but it is also internally consistent and it conforms to the more general view of Rascal as a one stop shop [19]. The only way to properly integrate extractors into this infrastructure was to re-engineer them from Python, ASF, Java and other languages to Rascal — mostly this was a manual re-engineering project. For XSLT the approach was semi-automated: the investment was justified by the larger number of scripts. The conceptual proximity of metalanguages also warranted closer investigation.

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).

2 Corpus

The following 43 files form our corpus, listed and linked here for possible replication purposes. One file, shared/xsl/xhtml2fo.xslt, was excluded as an outlier (big program from a domain of document typesetting, written by different people) to avoid distorting our data. The number of lines per file is given in brackets.

- shared/xsl/bgf-format.xslt (191)
- topics/investigation/analysis/bgf-overview.xslt (35)
- shared/xsl/bgf2bnf.xslt (209)
- topics/export/ebnf/bgf2dms.xslt (175)
- shared/xsl/bgf2dot.xslt (34)
- topics/export/hypertext/bgf2fancy.xslt (287)
- topics/export/rascal/bgf2rsc-unsafe.xslt (192)
- topics/export/rascal/bgf2rsc.xslt (228)
- shared/xsl/bgf2sdf.xslt (203)
- shared/xsl/bgf2tex.xslt (159)
- topics/export/txl/bgf2txl.xslt (167)
- topics/export/hypertext/bgf2xhtml.xslt (254)
- shared/xsl/btf2btf.xslt (22)
- topics/export/source/btf2source.xslt (119)
- shared/xsl/cbgf-split.xslt (9)
- shared/xsl/cbgf2cbgf-context.xslt (165)
- shared/xsl/cbgf2cbgf2cbgf.xslt (46)
- shared/xsl/cbgf2cbnf.xslt (101)
- shared/xsl/cbgf2xbgf-forward.xslt (332)
- shared/xsl/cbgf2xbgf-reverse.xslt (359)
- topics/extraction/w3c/cleanup.xslt (115)
- topics/extraction/ecore/ecore2bgf.xslt (531)
- shared/xsl/edd-export.xslt (24)
- shared/xsl/edd2dcg.xslt (261)
- shared/xsl/exbgf2xbgf.xslt (1828)
- topics/transformation/normalization/subdefs/extract.xslt (37)
- topics/mutation/expose-root/generate.xslt (21)
- topics/transformation/normalization/subdefs/inline.xslt (15)
- topics/extraction/ldf/ldf2bgf.xslt (14)
- shared/xsl/ldf2tex.xslt (571)
- topics/export/hypertext/ldf2xhtml.xslt (777)
- shared/xsl/mathml2tex.xslt (41)
- topics/export/hypertext/mathml2xhtml.xslt (42)
- topics/transformation/normalization/postfix2confix.xslt (73)
- topics/extraction/relax/rng2bgf.xslt (405)
- shared/xsl/rootprods.xslt (16)
- topics/extraction/w3c/spec2bgf.xslt (120)
- topics/extraction/w3c/spec2ldf.xslt (450)
- topics/extraction/txl/txl2bgf.xslt (296)
- shared/xsl/xbgf2cbgf.xslt (269)
- shared/xsl/xbgf2xbnf.xslt (277)
- topics/export/hypertext/xbgf2xhtml.xslt (410)
- shared/xsl/zoo2tex.xslt (104)

3 Metaprogramming Concepts of XSLT

XSLT [18], eXtensible Stylesheet Language Transformations, is a widely applicable XML manipulation language and as such it provides a wide selection of features. We will classify the concepts related to metaprogramming in eight categories:

- ◇ purely textual output and messages (§ 3.1),
- ◇ various kinds of templates (§ 3.2),
- ◇ template calls (§ 3.3),
- ◇ functions shared with XPath and XQuery (§ 3.4),
- ◇ conditionals and branching (§ 3.5),
- ◇ node targeting and context awareness (§ 3.6),
- ◇ copying and collecting values of nodes (§ 3.7) and
- ◇ element construction (§ 3.8).

3.1 Textual Output

Since some of the scripts in our corpus were related to pretty-printing, reformatting and benchmarking of XML structured data, there was quite a few concepts related to generation of purely textual content:

Terminal symbol (coded 315 times) is a common concept from grammarware: it means a string literal that is hard-coded into the grammar of the software language. The name "terminal" was given thanks to the fact that in context-free grammars reaching such a symbol means termination of the derivation process. We marked only those textual chunks as terminals that contained at least one proper word.

Non-alphanumeric character (403) is a special case of a terminal symbol that consists of one or few non-alphanumeric symbols: mostly quotes and brackets.

Space (44) was also coded separately because of its prominent occurrence.

Newline (91) was used even more commonly than a space because it was applicable to both text-targeting pretty-printers and XML-targeting transformers (in the latter case, to facilitate debugging).

Message (88) is a special XSLT feature allowing programs to print something on the screen without disturbing the data flow of the mapping — in our corpus it was mostly used for logging.

Error (14) is a pattern that was marked when an obviously erroneous input was dealt with by either producing a corresponding message or just capturing an `otherwise` case in a `choose` construct that had already covered all necessary sensible cases with `when` clauses.

Comment (1) is a standard XSLT feature to provide documentation — it was not used as actively as it could have been, perhaps because corresponding publications could be considered as proper documentation for the implemented grammar manipulation techniques, and the artefacts themselves were not considered essential.

3.2 Templates and Applications

In XSLT, functions are organised in "templates": code fragments that match a specified pattern in the input and rewrite it to the output pattern which is well-formed, but may combine both XSLT rewriting elements and target language elements. Templates can be also named to be explicitly called later. In total there were 611 templates identified in the corpus: 534 matched and 77 named. Their application was guided by 363 `apply-templates` instances and 428 direct calls (for the latter see the next subsection).

Universally matching templates (4) were relatively rare — they were matching `"*"` or some variation thereof limited to an unsupported namespace.

Root matching (41) — most scripts contained at least one special template that was meant to match the root element of the input and direct further rewriting to its elements. Just one file contained two root matches: obviously the grammar extractor from Ecore metamodels was happy to accept either `ecore:EPackage` or `xmi:XMI` since their content would have been identically structured anyway.

Match and map to singleton (125) was our code for templates which output consisted of a single textual construct or a single node either embracing simple content or delegating inner elements creation elsewhere.

Match and wrap in terminals (82) was needed to be coded separately due to its provenance: this pattern occurred often in both pretty-printing scripts and structural transformation scripts. In the latter case, wrapping terminals were often quotes.

Match and reapply to all children (10) did not occur that often, partly due to the fact that it could have been avoided altogether thanks to XSLT rewriting semantics.

Other non-parametric templates (368) included match-and-rewrite strategies of all kinds.

Other parametric templates (76) covered named rewritings defined once and called from appropriate places.

Apply templates to all children (215) is our code for `<xsl:apply-templates select="*"/>` or some variations thereof, it was quite a prominent way in XSLT to direct input traversals in a low-level fine-grained way available only in metaprogramming frameworks like Nuthatch [2].

Apply templates to one child (106) is the annotation used for more selective template applications. Due to the nature of XSLT, we can almost never be sure such constructs apply templates to just one child, it could be several siblings sharing the same name and

other selection criteria.

Apply templates to remote nodes (10) was a less common way to apply templates to something stored in a variable or in a parameter; usually as a means to reinstate normal matching traversals inside a called template.

Apply templates in the scope (37) — obviously, an `<xsl:apply-templates select="."/>` construct was seen only in a loop inside a selection operator; otherwise it would cause non-termination. It was used in cases when a simple XPath expression could not specify precisely the desired execution path.

3.3 Template Calls

There is more than one way to call a template: we have identified at least 7.

Empty call (2) is the way to use a static template that does not need any parameters to run. It was not very popular.

Call with yourself as a parameter (94) was quite popular: instead of relying on the scope of the rewriting being always available through `"."`, we provide it explicitly to the called template. This might seem counter-intuitive, but in fact implicitly enables several constructs for reliable referencing (since `"."` is context-specific, it will change its meaning inside loops).

Call with a child as a parameter (65) is a variation thereof: instead of providing the entire node, it is possible to propagate some of its children. In at least 3 cases of 65 coded such calls were associating a list of children to one template parameter.

Call with a variable as a parameter (42) was used mostly in the context of a system of several interdependent templates so that one would need to propagate some of its parameters or variables to the next one in the pipeline.

Recursive call (6) is self-descriptive and refers to a call to the same template where the call is performed.

Call with versatile parameters (7) was our desperate code when all of the above failed to properly describe a construct. These were the most sophisticated and often involved providing itself, a non-trivial selection of descendants and a complex expression extracting nodes of interest from a variable.

3.4 Built-in Functions

XPath, XQuery and XSLT use the same library of standard functions one can use to operate on XML nodes. Since SLPS used John Fleck's `xsltproc` [12] based on Daniel Veillard's `libxslt` [31], the selection of the available functions was limited to XSLT 1.0, and even then only the following 10 were used:

- ◇ `concat` (10), string concatenation
- ◇ `contains` (6), substring search
- ◇ `translate` (9), map strings on a per-character basis
- ◇ `string-length` (4), count the number of characters in a string
- ◇ `count` (22), count the number of nodes in a selected set (see also § 3.5)
- ◇ `substring` (78), `substring-after` (38) and `substring-before` (8), perform string carving, often used in combinations
- ◇ `local-name` (80), request the tag name of a node
- ◇ `normalize-space` (1), collapse whitespace in a string

3.5 Branching

Now we will consider 10 concepts related to conditional rewriting.

Check for existence (222) was the most popular way to determine the flow of rewritings. In 3 additional cases a `count` was performed and compared to zero instead — in functional programming this is considered a malpractice, but XSLT is not truly a lazily evaluated language.

Check if counter is exactly one (21) is another relatively popular practice to ensure that a selection criterion returned one and only one node.

Check if counter is positive (8) is performed in two similar circumstances: for checking that a selection criterion returned something (`count()>0`) or for checking that it returned a sequence of nodes (`count()>1`).

Check the local name of a node (162) is the second most popular code in this category. XSLT makes it easy to get a positive match of nodes without knowing their exact type (tag), so if treatment of some nodes diverges, such checks are necessary. For disjunctively combined checks we treated them as several separate ones.

Check for emptiness (6) entails comparing a calculated value to a `''`.

Compare with a hard-coded value (122) is a generalisation of that, covering comparisons to predefined string and integer values for cases that were not classified with codes already described above.

Emulate if-then-else (111) with a `choose`, `when` and `otherwise` is a popular hack. One of the known shortcomings of XSLT is the lack of two-branched conditional statement present in many other software languages: there is a one-branched `if` that performs something if a condition holds and does nothing otherwise, and there is a multi-branched `choose`. Out of 378 uses of `choose` within our corpus, almost one third has only two clauses and looks like an if-then-else con-

struct. This prevalence is on par with one-branched `if` (138 times).

Empty branch (24) is another XML-specific idiom. 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 proper language workbenches like Rascal with the classic functional programming trick of a "poor man's Maybe" (a list which is either empty or a singleton) and inlining.

Variable declaration with multiple possible values (6) is a somewhat uncommon pattern that was coded because it was not anticipated. Usually a variable is declared with a formula to calculate its value directly — however, on several occasions there was no formula, but instead a complex content fragment with non-trivial branching and fetching.

3.6 Node Targeting

The following 16 concepts that we put into this category, deal with ways of accessing target nodes according to their context.

Global stylesheet parameters (16) were used to provide additional inputs (usually timestamps and filenames to be used as additional inputs) with `--param` and `--stringparam` options of `xsltproc` [12].

Accessing external documents (10) through such parameters is usually done in three steps: a parameter is declared; an extra variable is declared using `document()` function to use that parameter to access a document; templates from anywhere within the stylesheet get access to an extra document (or its part) through that global variable.

Deep match (33) uses two slashes (`//`) instead of one and matches any descendant node instead of immediate children.

Multimatch (40) happens when a template or any selection construct must match more than one pattern, they are just joined with a vertical bar: `a|b` matches a set of nodes each of which is either `a` or a `b`. We paid a lot of attention to such a feature because its presence is not guaranteed in other metaprogramming languages (e.g., Rascal does not have it yet).

Next sibling (2) can be obtained in XSLT by asking the parent node directly; again, manipulating parent nodes is not an omnipresent functionality in metaprogramming frameworks.

Head and tail (50) is an awkwardly looking yet frequently used pattern in XSLT when something is applied to the first element of a particular selection set (say, `x[1]`) and then a clone of the same code is repeatedly applied to the remainder of the same set (say, `x[position()>1]`).

Loop over nodes (150) is a code to cover all kinds of other `for-each` loops. Only in 1 case of those the selection collection was sorted by position in descending order.

Position (first: 3; last: 2; fixed: 6) is handled directly on several occasions and is used to assign special behaviour to the first or last elements in a collection of those matching the selection criterion; arbitrary positions were used in analysis and explicit pretty-printing or unique IDs.

Child by number (first: 84, second: 34, third: 1) can be also used explicitly, as in `a[1]`, `a[2]`, etc. From the collected numbers we see that the first child pattern is the most common one — sometimes it was also used as an extra safeguard to ensure that the selection result has exactly one node. In many proper high level metaprogramming frameworks this scenario would be improved by assigning names to the first and the second children if their number is known a priori.

Uniqueness (12) checking is implemented in XSLT 1.0 with a trick well known to any experienced XSLT programmer, it looks approximately like this: `count(/x/y[not(text()=../preceding-sibling::y/text())])`, sometimes with a deep match, if appropriate.

Target through local names (26) is a hack with multiple purposes, its essence is the fact that one uses a construct like `*[local-name()="x"]` instead of a more transparent `x` pattern. First, it allows to combine the entire matching logic in one place. Second, it relaxes namespace constraints (`*:x` could work, but will not match the default namespace). Third, it provides easy means of traversing a complement (by flipping the equality condition to inequality).

3.7 Copies and Values

Metaprogramming of the kind we discuss in this paper, is founded on three main principles: pattern matching and rewriting strategies being only the first obvious two of them. The third principle is the ability to reuse fragments of input in creation of the output. This section contains several popular concepts concerning such fragment reuse.

Copy structured node (289) is our code for using copying instead of taking a value of a node. In XSLT, `value-of` can be used on any element, and for composite elements it produces their purely textual representation, brutally concatenated if several nodes match the selection criterion (a rough Rascal equivalent would be string splicing). However, in the real code this option has never been used: all structured elements were copied (289 times), and only textual ones were taken by values (537 times). In the remainder of this section there is no distinction made between them.

Value of the scope node (132) is simply `<xsl:value-of select="."/>`.

Value of a local (descendant) node (386) is an even more popular variant when a child or a descendant node is taken the value of.

Value of a parent (ancestor) node (12) is a separate case when the value is taken from a parent, a sibling or any other node that can only be reached by walking up a tree.

Value of a remote node (146) covers `value-of`s with parameters or variables used in the selection criteria; those were mostly used inside named templates.

Value of a counter (8) codes the usage of `count()` on some XPath expression.

Scope shielding (34) happens when we need to construct a complicated selection expression for which the current scope should be preserved. We have briefly discussed `"."` being context-specific in § 3.3. Consider a situation when we need to look up a person by its ID, it would probably look not unlike `$people/*[id=$id]/name`, for which we need to preserve the current ID in scope in a variable in order to be able to express that *that* ID which will be local during the future matching, should be equal to *this* ID that we see in scope now. It is a well-known trick, but unlike others that map to proper language constructs in other metaprogramming languages, this is an idiosyncratic consequence of the non-unifying declarative paradigm with ad-hoc defined variables adopted by XSLT.

3.8 Construction

Pure construction of output nodes should also be possible, and we have five concepts belonging to that category.

Node construction (454) was coded only for top elements, otherwise its count would be far into thousands: it covers any inline construction of an output node in HTML, BGF, LDF or whatever the output dialect of XML is.

Explicit element construction (4) was used only on several occasions. It looks like an `<xsl:element>` tag and differs from normal node construction in the only way: its tag name can be evaluated on the fly in a construct of any complexity.

Explicit attribute construction (12) is a similar feature for constructing attributes for output tags.

In-place evaluation (21) is a shorthand notation for the previous construct. For instance, `<a href="{link}"/>x</a>` treats `link` as an expression to be evaluated (in this case — the immediate child of this name, flattened to its text) as if we had written `<a> <xsl:attribute name="href" > <xsl:value-of select="link"/> </xsl:attribute> <xsl:text>x</xsl:text> </a>`. All such curly bracket uses were coded for in-place evaluation.

Empty text node (29) produces a textual output which is empty.

4 Preliminary Analysis

As should be apparent from the data we have collected and presented in the previous section, most XSLT features used in the scope of our metaprogramming activities, can be mapped directly to any proper metaprogramming/language workbench, including Rascal — the one we intended to use in this project. During the case studies on the existing XSLT-based grammar extractors, we found out that the following metaprogramming idioms are particularly easy to express in either XSLT or 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 harder to match:

- 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.
- Multimatches — see § 3.6.
- 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.

5 Related Work

Manipulating XML-based encodings has been acknowledged as a metaprogramming activity for more than a decade [29]. However, XSLT does not have to be seen as a metaprogramming language — it has been used for anything from requirements verification [10] to (XMI-based) model transformation [16] or as a host language for embedding semantic web query languages [14]. However, in this paper we view it strictly as a language that can express structural patterns and transform them to other structural patterns.

Starting with qualitative data, annotating it with codes and grouping them into concepts and concepts in turn into categories, if done properly, is called *grounded theory*. It is a relatively new approach to see in software evolution research, but there has been some recent work demonstrating its applicability and usefulness in many topics from interface design [25] to software architecture [32], as well as in identifying open problems [24]. Not wishing to join the debate of what it truly is and what form should it take for computer science, we did not position this project as a grounded theory endeavour, but did borrow some of the terminology whenever it seemed appropriate. In any case, this paper is definitely a report on qualitative data/code analysis.

Migrating code from an untyped language to a typed language has never been easy but is well-researched in the form of *type inference* techniques, also for declarative languages [1, 27] as well as for the xmlware technological space [7, 23]. The current trend in type inference is to use as much contextual information as possible, mining any data available in the ecosystem [26] such as test suites [40]. We must confess that the presence of comprehensive test data collections would have made the migration much more comfortable.

Researching the evolution of a software language implementation is not a new topic either, it has very strong representatives [30]. Essentially our situation is a *coupled evolution* scenario where the changes in the metalanguage should co-occur with the changes to the mappings expressed in it. This is a hard open problem without a solid convincing solution. We are only aware of quite modest and therefore realistic migration case studies from XSLT to XQuery [5] or even within XSLT from one XSD to another [33]. The current differences among metalanguages and their environments do not indicate any complexity drops of this problem in the near future [11].

Alternative routes included using formalisations of XSLT [6] which is a tempting idea since it can possibly produce fully automated provably semantic-preserving metatransformations — *if* the formalisations are expressive enough. This direction seems especially promising due to recent advances in type checking of XSLT transformations [23, 28].

This project was essentially code migration from XSLT [12, 18] to Rascal [20, 21]. Both the source and the target frameworks were open source, and the systems were of a quite modest size, so the heaviest problems in practical code migration projects did not manifest themselves, ultimately enabling the success of this project.

6 Conclusion

Motivated by a case study of migrating metaprogramming scripts from XSLT to Rascal, we have investigated 10+ KLOC of XSLT stylesheets and presented several categories of metaprogramming concepts specific to XSLT. This analysis reflects how XSLT was used in the metaprogramming context and serves as a foundation for spotting differences and shortcomings of alternative frameworks.

References

[1] H. Azzoune. Type Inference in Prolog. In E. L. Lusk and R. A. Overbeek, editors, *Proceedings of the Ninth International Conference on Automated Deduction*, volume 310 of *LNCS*, pages 258–277. Springer, 1988.

[2] A. H. Bagge and R. Lämmel. Walk Your Tree Any Way You Want. In K. Duddy and G. Kappel, editors, *Proceedings of the Sixth International Conference on Theory and Practice of Model Transformations*, volume 7909 of *LNCS*, pages 33–49. Springer, 2013.

[3] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke. Automated Software Transplantation. In *Proceedings of the 24th International Symposium on Software Testing and Analysis*, pages 257–269. ACM, 2015.

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

[5] R. Bettentrupp, S. Groppe, J. Groppe, S. Böttcher, and L. Gruenwald. A Prototype for Translating XSLT into XQuery. In Y. Manolopoulos, J. Filipe, P. Constantopoulos, and J. Cordeiro, editors, *Proceedings of the Eighth International Conference on Enterprise Information Systems: Databases and Information Systems Integration (ICEIS/DISI)*, pages 22–29, 2006.

[6] G. J. Bex, S. Maneth, and F. Neven. A Formal Model for an Expressive Fragment of XSLT. In *Proceedings of the First International Conference on Computational Logic*, volume 1861 of *LNCS*, pages 1137–1151. Springer, 2000.

[7] D. Colazzo and C. Sartiani. Precision and Complexity of XQuery Type Inference. In *Proceedings of the 13th International Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 89–100. ACM, 2011.

[8] J. R. Cordy. The TXL Source Transformation Language. *Science of Computer Programming*, 61(3):190–210, 2006.

[9] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley, 2000.

[10] A. Durán, A. R. Cortés, R. Corchuelo, and M. Toro. Supporting Requirements Verification Using XSLT. In *Proceedings of the 10th Anniversary Joint International Requirements Engineering Conference*, pages 165–172. IEEE Computer Society, 2002.

[11] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning. The State of the Art in Language Workbenches — Conclusions from the Language Workbench Challenge. In M. Erwig, R. F. Paige, and E. Van Wyk, editors, *Proceedings of the Sixth International Conference on Software Language Engineering*, volume 8225 of *LNCS*, pages 197–217. Springer, 2013.

[12] J. Fleck. `xsltproc` — command line XSLT processor. http://linux.die.net/man/1/xsltproc, 2001.

[13] T. Gîrba, J.-M. Favre, and S. Ducasse. Using Meta-Model Transformation to Model Software Evolution. *Proceedings of the Second International Workshop on Metamodels, Schemas and Grammars for Reverse Engineering (ateM)*, 137(3):57–64, 2005.

[14] S. Groppe, J. Groppe, V. Linnemann, D. Kukulenz, N. Hoeller, and C. Reinke. Embedding SPARQL into XQuery/XSLT. In R. L. Wainwright and H. Haddad, editors, *Proceedings of the 23rd Symposium on Applied Computing*, pages 2271–2278. ACM, 2008.

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

[16] H. Gustavsson, B. Lings, B. Lundell, A. Mattsson, and M. Beekveld. Simplifying Maintenance by using XSLT to Unlock UML Models in a Distributed Development Environment. In *Proceedings of the 23rd International Conference on Software Maintenance*, pages 465–468. IEEE, 2007.

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

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

[19] 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, *Revised Selected Papers of the Fourth International Conference on Software Language Engineering*, volume 6940 of *LNCS*, pages 1–18. Springer, 2011.

[20] 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 *LNCS*, pages 222–289. Springer, 2009.

[21] P. Klint, T. van der Storm, and J. J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the Ninth International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 168–177. IEEE Computer Society, 2009.

[22] P. Klint, J. 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.

[23] M. Lepper and B. Trancón y Widemann. A Simple and Efficient Step Towards Type-Correct XSLT Transformations. In *Proceedings of the 26th International Conference on Rewriting Techniques and Applications*, volume 36 of *LIPIcs*, pages 350–364. Schloss Dagstuhl, 2015.

[24] E. Mustonen-Ollila, H. Nyerwanire, and A. Valpas. Knowledge Management Problems in Healthcare — A Case Study based on the Grounded Theory. In *Proceedings of the International Conference on Knowledge Management and Information Sharing*, pages 15–26. SciTePress, 2014.

[25] C. Rivers, J. Calic, and A. Tan. Combining Activity Theory and Grounded Theory for the Design of Collaborative Interfaces. In *Proceedings of the First International Conference on Human Centered Design*, volume 5619 of *LNCS*, pages 312–321. Springer, 2009.

[26] B. Spasojevic, M. Lungu, and O. Nierstrasz. Mining the Ecosystem to Improve Type Inference for Dynamically Typed Languages. In *Proceedings of the Fourth Symposium on New Ideas in Programming and Reflections on Software (Onward!)*, pages 133–142. ACM, 2014.

[27] P. Tarau. On Logic Programming Representations of λ Terms: de Bruijn Indices, Compression, Type Inference, Combinatorial Generation, Normalization. In E. Pontelli and T. C. Son, editors, *Proceedings of the 17th International Symposium on Practical Aspects of Declarative Languages (PADL)*, volume 9131 of *LNCS*, pages 115–131. Springer, 2015.

[28] A. Tozawa. Towards Static Type Checking for XSLT. In *Proceedings of the First Symposium on Document Engineering (DocEng)*, pages 18–27. ACM, 2001.

[29] B. Trancón y Widemann, M. Lepper, and J. Wieland. Automatic Construction of XML-Based Tools Seen as Meta-Programming. *Automated Software Engineering*, 10(1):23–38, 2003.

[30] L. Tratt. Evolving a DSL Implementation. In R. Lämmel, J. Visser, and J. Saraiva, editors, *Revised Papers of the Second International Summer School on Generative and Transformational Techniques in Software Engineering*, volume 5235 of *LNCS*, pages 425–441. Springer, 2007.

[31] D. Veillard. `libxslt` — library used to do XSL transformations on XML documents. http://xmlsoft.org/XSLT/, 2001.

[32] M. Waterman, J. Noble, and G. Allan. How Much Up-Front? A Grounded Theory of Agile Architecture. In *Proceedings of the 37th International Conference on Software Engineering (ICSE), Volume 1*, pages 347–357. IEEE, 2015.

[33] Y. Wu and N. Suzuki. Detecting XSLT Rules Affected by Schema Evolution. In C. Vanoirbeek and P. Genevès, editors, *Proceedings of the 15th Symposium on Document Engineering (DocEng)*, pages 143–146. ACM, 2015.

[34] V. Zaytsev. Language Evolution, Metasyntactically. *Electronic Communications of the European Association of Software Science and Technology (EC-EASST); Bidirectional Transformations*, 49, 2012.

[35] V. Zaytsev. Notation-Parametric Grammar Recovery. In A. Sloane and S. Andova, editors, *Post-proceedings of the 12th International Workshop on Language Descriptions, Tools, and Applications (LDTA 2012)*. ACM Digital Library, June 2012.

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

[37] V. Zaytsev. Case Studies in Bidirectionalisation. In *Pre-proceedings of the 15th International Symposium on Trends in Functional Programming (TFP 2014)*, pages 51–58, May 2014. Extended Abstract.

[38] V. Zaytsev and A. H. Bagge. Parsing in a Broad Sense. In J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfrán, editors, *Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 8767 of *LNCS*, pages 50–67. Springer, 2014.

[39] 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.

[40] H. Zhu, A. V. Nori, and S. Jagannathan. Dependent Array Type Inference from Tests. In D. D'Souza, A. Lal, and K. G. Larsen, editors, *Proceedings of the 16th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 8931 of *LNCS*, pages 412–430. Springer, 2015.

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

```
<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.