Evolution of Metaprograms

Dr. Vadim Zaytsev aka @grammarware
SATToSE 2015
Languages, Models and Megamodels
A Tutorial

Anya Helene Bagge¹ and Vadim Zaytsev²

¹ BLDL, University of Bergen, Norway, anya@ii.uib.no
² University of Amsterdam, The Netherlands, vadim@grammarware.net

Abstract. We all use software modelling in some sense, often without using this term. We also tend to use increasingly sophisticated software languages to express our design and implementation intentions towards the machine and towards our peers. We also occasionally engage in metamodelling as a process of shaping the language of interest, and in megamodelling as an activity of positioning models of various kinds with respect to one another.

This paper is an attempt to provide a gentle introduction to modelling the linguistic side of software evolution; some advanced users of modelware will find most of it rather pedestrian. Here we provide a summary of the interactive tutorial, explain the basic terminology and provide enough references to get one started as a software linguist and/or a megamodeller.

1 Introduction

This paper is intended to serve as very introductory material into models, languages and their part in software evolution — in short, it has the same role as the tutorial itself. However, the tutorial was interactive, yet the paper is not: readers familiar with certain subtopics would have to go faster through certain sections or skip them over.

In §2, we talk about languages in general and languages in software engineering. In §3, we move towards models as simplifications of software systems. The subsections of §4 slowly explain megamodelling and different flavours of it. The tutorial paper is concluded by §5.

2 Software Linguistics

Let us start by examining what a language is in a software context.

In Wikipedia, the concept is described³ as follows:

Language is the human ability to acquire and use complex systems of communication, and a language is any specific example of such a system. The scientific study of language is called linguistics.

³ http://en.wikipedia.org/wiki/Language
Evolution!
Meta program
Metaprogram
Metaprogramming

Grammar

Language

Grammar'

Script

Mapping

Language'

Program

MapApply

Program'

definitionOf

definitionOf

elementOf

elementOf

elementOf

realisationOf

inputOf

hasOutput
Examples
Example: ANTLR→BGF

```
grmmarDef:
{
    try{
        DocumentBuilderFactory dbfac = DocumentBuilderFactory.newInstance();
        DocumentBuilder docBuilder = dbfac.newDocumentBuilder();
        doc = docBuilder.newDocument();
        root = doc.createElement("bgf:grammar");
        doc.appendChild(root);
    }catch (Exception e){System.out.println(e);}
}

'grammar' ID ';' NEWLINE rule+
{
    try{
        Transformer trans = TransformerFactory.newInstance().newTransformer();
        trans.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "no");
        trans.setOutputProperty(OutputKeys.INDENT, "yes");
        trans.transform(new DOMSource(doc), new StreamResult(new
            FileOutputStream(output)));
    }catch (Exception e){System.out.println(e);}
};
```
Example: SDF→BGF

equations

[map-sdf-definition-to-bgf]
&C*1 := trafoProds (accuProds (&M*1,))

main (definition &M*1)

= <bgf:grammar xmlns:bgf="http://planet-sl.org/bgf">
   &C*1
</bgf:grammar>

[map-sdf-module-to-bgf]
&C*1 := trafoProds (accuProds (&M1,))

main ( &M1 )

= <bgf:grammar xmlns:bgf="http://planet-sl.org/bgf">
   &C*1
</bgf:grammar>

[exclude-lexical-productions]
accuProds ( lexical syntax &Ps1, &P*1 ) = &P*1
for (Class<?> clss : classes) {
    Element rule = doc.createElement("bgf:production");
    Element nonterminal = doc.createElement("nonterminal");
    Element rhs = doc.createElement("bgf:expression");
    nonterminal.appendChild(doc.createTextNode(clss.getSimpleName()));
    root.appendChild(rule);
    rule.appendChild(nonterminal);
    rule.appendChild(rhs);
    Collection<Element> tmp = new LinkedList<Element>();
    if (clss.isEnum()) {
        compoisor = "choice";
        unit = "empty";
        for (Object c : clss.getEnumConstants()) {
            Element selectable = doc.createElement("selectable");
            tmp.add(selectable);
            Element selector = doc.createElement("selector");
            selectable.appendChild(selector);
            selector.appendChild(doc.createTextNode(c.toString()));
            Element expr = doc.createElement("bgf:expression");
            selectable.appendChild(expr);
            Element empty = doc.createElement("epsilon");
            expr.appendChild(empty);
        }
    }
} else if ...
Example: LDF→BGF

```xml
<?xml version="1.0"?>
    xmlns:ldf="http://planet-sl.org/ldf" version="1.0">
    <xsl:output method="xml" encoding="UTF-8"/>
    <xsl:template match="/ldf:document">
        <bgf:grammar>
            <xsl:apply-templates select="/bgf:production"/>
        </bgf:grammar>
    </xsl:template>
    <xsl:template match="bgf:production">
        <xsl:if test="local-name(../..) != 'example'">
            <xsl:copy-of select="."/>
        </xsl:if>
    </xsl:template>
</xsl:stylesheet>
```

SLPS/topics/extraction/ldf/ldf2bgf.xslt
Example: TXL→BGF

```xml
<xsl:template match="literalOrType">
  <xsl:choose>
    <xsl:when test="type/typeSpec/typeid/literal/unquotedLiteral/special='!'">
    </xsl:when>
    <xsl:when test="type/typeSpec/opt_typeModifier/typeModifier='see'">
    </xsl:when>
    <xsl:when test="type/typeSpec/opt_typeRepeater/typeRepeater='+'">
      <bgf:expression>
        <plus>
          <bgf:expression>
            <nonterminal>
              <xsl:value-of select="type/typeSpec/typeid/id"/>
            </nonterminal>
          </bgf:expression>
        </plus>
      </bgf:expression>
    </xsl:when>
    <xsl:when test="type/typeSpec/opt_typeRepeater/typeRepeater='*' or type/typeSpec/opt_typeModifier/typeModifier='repeat'">
      <bgf:expression>
        <star>
          <xsl:if test="type/typeSpec/typeid/id">
            <nonterminal>
              <xsl:value-of select="type/typeSpec/typeid/id"/>
            </nonterminal>
          </xsl:if>
        </star>
      </bgf:expression>
    </xsl:when>
  </xsl:choose>
</xsl:template>
```

SLPS/topics/extraction/txl/txl2bgf.xslt
Example: XMI→BGF

```xml
<xsl:template match="/ecore:EPackage">
  <bgf:grammar>
    <xsl:apply-templates select="*"/>
  </bgf:grammar>
</xsl:template>

<xsl:when test="@xsi:type='ecore:EEnum'">
  <bgf:production>
    <nonterminal>
      <xsl:value-of select="$ourName"/>
    </nonterminal>
    <xsl:choose>
      <xsl:when test="count(eLiterals)=0">
        <bgf:expression>
          <epsilon/>
        </bgf:expression>
      </xsl:when>
      <xsl:when test="count(eLiterals)=1">
        <xsl:apply-templates select="./eLiterals"/>
      </xsl:when>
      <xsl:otherwise>
        ...
      </xsl:otherwise>
    </xsl:choose>
  </bgf:production>
</xsl:when>
```

SLPS/topics/extraction/ecore/ecore2bgf.xslt
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 programs, analysing, parsing, transforming, 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.

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 artifact 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.

http://grammarware.net/writes/#XSLT-to-Rascal2015
TL;DR

- Useless preliminary analysis
- XSLT is declarative
- Rascal is functional/imperative
- XSLT is XML-like
- Rascal is Java-like
TL;DR

- Translates well
  - named callables
  - pattern matching
  - condition checking
  - comprehensions
TL;DR

• Problems
  • XSLT is untyped: `<xsl:when test="..." />`
  • Multipattern matching: a|b
  • Vectors vs scalars
  • No variables in λ
Questions

- Does anyone recognise the problem?
- Has anyone ever done anything alike?
- Are there good solutions?
- Why is life so hard?

http://grammarware.github.io