Electronic Communications of the EASST
Volume 65 (2014)

Proceedings of the International Workshop on Software Quality and Maintainability (SQM 2014)

Software Language Engineering by Intentional Rewriting

Vadim Zaytsev

17 pages

Guest Editors: Lodewijk Bergmans, Tom Mens, Steven Raemaekers
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Software Language Engineering by Intentional Rewriting

Vadim Zaytsev

Universiteit van Amsterdam, The Netherlands, vadim@grammarware.net

Abstract: Grammars in a broad sense (specifications of structural commitments) are complex artefacts that define software languages. Assessing and improving their quality in an automated, non-idiosyncratic manner is an unsolved problem which we face in an especially acute form in the case of mass maintenance of hundreds of heterogeneous grammars (parser specs, ADTs, metamodels, XML schemata, etc) in the Grammar Zoo. In an attempt to apply software language engineering methods to solve a software language engineering problem, we design a language for *grammar mutations* capable of applying uniform intentional transformations in the scope of a big grammar or a corpus of grammars. In this paper, we describe a disciplined process of engineering such a language by systematic reuse of semantic components of another existing software language. The constructs of the reference language are analysed and classified by their intent, each category of constructs is then subjected to rewriting. This process results in a set of constructs that form the new language.

Keywords: term rewriting; intentionality; grammar programming; software language engineering; grammar mutation; grammarware.

1 Introduction

Although there have been a lot of expert opinions expressed about designing a software language [vW65, Hoa73, Wir74, MHS05, VBD+13], the process often remains far from being completely controlled, and the correspondence of language design decisions with the successful uses of the language for intended tasks, remains unproven. Formalising domain knowledge and expressing it algorithmically is what we see as one of the fundamental challenges that the field of software language engineering is facing.

Our case study concerns a domain-specific language for manipulating grammars in a broad sense — in fact, structural contracts like language concrete syntaxes or library interfaces [KLV05]. In earlier work, we have been continuously addressing the problem of expressing evolutionary changes to these structural contracts as transformation steps, showing the superiority of detail of such specifications to inline grammar editing [Läm01a, LZ09, LZ11]. We have also identified the need for expressing large scale manipulations — transformation generators [Zay11] or grammar mutations [Zay12b], cautiously proposing one or two as the practical side dictated.

In this paper, we are determined to construct a full-fledged language for large scale grammar programming, which would implement grammar mutations. If the language for fine-grained grammar programming had operators like “rename this nonterminal” or “eliminate this unused nonterminal”, then for the language of large scale grammar programming, we aim to have commands like “rename all nonterminals to lowercase” and “eliminate all unused nonterminals”. In order to do so, we deconstruct the existing language and intentionally (as in “intentional soft-

ware" [SCC06]) generalise them. We automate the process of constructing this new software language, by systematic reuse and rewriting of the preconditions and the rewriting rules of the original transformation operators.

By applying such a technique to engineer a new language, we expect: (a) to define this new language, (b) to infer, possibly with some human supervision, its implementation, and (c) to validate the design of the base language by the possibility of its automated reuse.

Our contributions include, indeed, a new language, defined as **235** elementary grammar mutations that are composable into more complex ones; the implementation of this language within Rascal language workbench [KSV11], obtained by metaprogramming techniques and semi-automated rewriting; and we identified several *design flaws* in the base language. All deliverables are publicly available through GitHub: the experimental code can be observed in the SLPS repository at http://github.com/grammarware/slps, and after sufficient testing and documenting it will be added to the GrammarLab library of Rascal at http://github.com/cwi-swat/grammarlab. Overall, the experiment was successful, and this extensive report on it could be of use for next steps of research on automated software language engineering.

2 Problem details

In [LZ09, LZ11], we have proposed XBGF[1], a software language for *grammar programming*, which is a method of imposing systematic changes to a grammar by parametric use of transformation operators [KLV05, Läm01a, DCMS02], as opposed to *grammar hacking*, which entails untraceable inline editing of the grammar at hand [Läm01b]. In fact, each of the operators that comprise the language, is a rewriting system that transforms a term algebraic representation of a formal grammar, according to its specification. For instance, the **unfold** operator replaces a nonterminal occurrence with its definition, the **renameN** renames a nonterminal, the **reroot** operator changes the starting symbol of the grammar, etc. The design of XBGF was validated by performing a major case study involving six grammars of Java [LZ11]. It was later extended for bidirectionality [Zay12b] and acquired several new operators by forming bidirectional pairs with existing ones. However, grammar programming with XBGF often feels rather low level, since it can take several transformation steps to address one issue, and the *intent* of using a particular operator is impossible to infer by looking at one step without considering its context. This makes both mass maintenance of grammars with XBGF transformations, and maintenance of the already programmed transformation scripts, more difficult than they could technically be.

We represent grammars as quadruples $\langle \mathbf{N}, \mathbf{T}, \mathbf{P}, \mathbf{S} \rangle$, where $\mathbf{N}$ is a finite set of nonterminal symbols, $\mathbf{T}$ is the finite set of terminal symbols, $\mathbf{P}$ is the set of production rules having a form of $n ::= x$, where $n \in \mathbf{N}$ is the nonterminal being defined, x is a defining algebraic expression over terminals and nonterminals, and $\mathbf{S} \subset \mathbf{N}$ is a set of starting nonterminal symbols[2]. We also assume that $\mathbf{N} \cap \mathbf{T} = \varnothing$ (by treating terminals and nonterminals separately). With $\mathbf{P}_n$ we will denote the subset of all production rules that define the nonterminal $n \in \mathbf{N}$: $\mathbf{P}_n = \{n ::= e_i\} \subset \mathbf{P}$.

Let us define a grammar transformation operator as a tuple $\tau = \langle \pi, \varphi \rangle$, where π is a precon-

[1] The language reference of XBGF is available at http://slps.github.io/xbgf.

[2] The actual GrammarLab implementation is slightly more complex and deals with labels and markers: we exclude all transformations and mutations concerning them for space considerations. http://grammarware.github.io/lab/

```
XResult runEliminate(str x, GGrammar g) {
  if (x in g.S) return <problemStr("Cannot eliminate root nonterminal",x),g>;
  if (x notin g.N) return <freshName(x),g>;
  GProdList ps = [p | GProd p <- g.P, p.lhs != x];
  if (/nonterminal(x) := ps) return <notFreshName(x),g>;
  return <ok(), grammar(g.N - x, ps, g.S)>; }

GGrammar EliminateTop(GGrammar g) {
 for (x <- g.N - g.S)
  {<ps1,_,ps3> = splitPbyW(g.P, innt(x));
  if (/nonterminal(x) !:= ps1+ps3) g.P = ps1 + ps3; }
 return g; }
```

Figure 1: *Top:* the **eliminate** operator implemented in Rascal — the implementation used in GrammarLab is already enhanced to cover both regular execution and negotiated one [Zay14c]. *Bottom:* the **EliminateTop** grammar mutation implemented in Rascal.

dition and φ is the rewriting rule[3]. Then, a grammar transformation will be τ_{a_i}, where a_i are its parameters. The type and quantity of parameters differ from one operator to another: for example, consider an **introduce** operator, $a_1 = [p_1; p_2; \ldots; p_n]$ is the list of production rules that need to be introduced into the grammar; π is asserting that all the provided production rules indeed define one nonterminal symbol, which is not present in the grammar; and φ is a trivial operation of adding all p_i to **P**.

As can be seen from Figure 1, this formalisation maps directly to the language implementation: either the straightfoward one or the one allowing the negotiated transformation model per [Zay14c], they just contain more details like error messages. An alternative Prolog implementation is more verbose but works similarly — we save space here by displaying only Rascal [KSV11] code.

A grammar mutation is defined differently: its precondition works not as an applicability condition, but as a trigger for possible application of φ, thus being a classic term rewriting system in itself. The implicit postcondition in such case is the conjunction of all negated preconditions: when *no* rewriting rules are applicable, the mutation has *successfully* terminated. Being described as general automation strategies, mutations are also allowed to be more complex, with one step covering multiple changes (typical for normalisations). Thus, we define a grammar mutation as $\mu = \langle \{\pi^{(i)}\}, \{\varphi^{(i)}\} \rangle$, with each $\pi^{(i)}$ being a trigger for applying the corresponding $\varphi^{(i)}$ rewriting strategy. An corresponding implementation code can be found on Figure 1.

Grammar mutations are not entirely a new subject. There was a need for fully automated grammar transformations expressed in the first grammar convergence project [LZ09], where we deployed so called *generators* that exercised more automation that regular grammar transformations [Zay11]. The name "generator" was used due to the fact that they were implemented as higher order functions, taking a grammar as an input and generating a sequence of transformation steps that would still need to be applied to it. Later it was deemed to be an implementation detail, and many currently used mutations are implemented differently, if a more efficient algorithm exists. An overview of available generators at that time can be found in [Zay10, §4.9].

[3] We deliberately exclude the postcondition from all the formulae in the current paper, because we will not base any substantial decisions on it.

Such strategies cannot be treated as transformation operators, since effectively they need the whole grammar to act as a parameter.

Grammar mutations in the form discussed here, were first mentioned in [Zay12b, §3] as a possibly bidirectional, but not bijective, mapping between grammars. In [Zay12c, §3.8.1] one can find an overview of many possible mutation algorithms. This paper is the first account of exploring the space of possible grammar mutations systematically, by basing them on controlled extensions of an existing operator suite. When a mutation μ is an intentionally generalised version of the transformation operator τ, we will say that $\tau \vdash \mu$.

3 Intentional generalisation

We have identified four different scenarios of successful generalisation, based on the intent of the target mutation. Mutations of Type I are trivial: they turn operators with *"check that π holds, then do φ"* semantics into mutations with *"for all cases where π holds, do φ"* behaviour. Mutations of Type II require additional information that can be automatically obtained by traversing the input grammar or performing any additional computation: a "reroot to top" is an illustrative example — we need to compute the set of top nonterminals, and then perform the usual **reroot**. Mutations of Type III generalise over operators with multiple intents: for instance, factoring can be done in many different ways, so the inference of mutations require classifying those intents and splitting them (thus, one operator may yield several mutations). Mutations of Type IV are parametric, they include grammar composition, slicing and other sophisticated manipulations. The following sections go deeper into details of obtained mutations, with the emphasis of identifying use cases.

The implementation of XBGF is rigidly structured (cf. Figure 1), which allowed for systematic semi-automated rewriting of the operator code in order to obtain the code of the corresponding grammar mutation(s).

3.1 Mutations of Type I: trivial generalisation

The transformation operators **abridge**, **deyaccify**, **distribute**, **eliminate**, **equate**, **fold**, **horizontal**, **inline**, **unchain**, **unfold** and **vertical** can be turned into mutations trivially: by attempting to apply them to any nonterminal (or any production rule, depending on the expected type of arguments) that is present in the grammar. For example, **unfold** will fail on any unused nonterminal and on any recursively defined one, and will succeed otherwise. A more optimal definition of these mutations is as follows.

Definition 1 (trivially generalisable mutation) Given an operator $\tau = \langle \pi, \varphi \rangle$ parametrised with a production rule, a production label, a nonterminal symbol or two nonterminal symbols, a trivially generalisable mutation based on that operator, has the form $\mu_{\mathrm{I}} = \langle \{\pi\}, \{\varphi\} \rangle$. ■

In other words, a trivially generalisable grammar mutation performs rewriting of the grammar for each of the components that satisfies the precondition of the base operator.

abridge $\vdash$ ***AbridgeAll:*** remove all reflexive chain production rules. Perform **abridge**(p) for all $p \in \mathbf{P}$, such that $p = x ::= x$, where $x \in \mathbf{N}$. Such mutation has no counterpart in prior work, but it is very reasonable: indeed, it is rarely the *intent* of the grammar engineer to remove some reflexive

chain production rules while retaining others — it is more usual to commit to a different grammar style where none are needed, which technically can still result in just one transformation step.

deyaccify $\vdash$ ***DeyaccifyAll:*** remove all yaccified production rules. Perform **deyaccify**(x) for all $x \in \mathbf{N}$, such that $\mathbf{P}_x = \{x ::= e_1, x ::= e_2\}$, where e_1 and e_2 follow one of yaccification patterns. A corresponding generator was proposed previously in [Zay10, §4.9] and [Zay11, §5.4].

A "yaccified" definition [Läm01a, JM01] is named after YACC [Joh75], a compiler compiler, the old versions of which required explicitly defined recursive nonterminals — i.e., in order to define a nonterminal x as y^+, one would in fact write $x ::= y$ and $x ::= xy$, because in LALR parsers like YACC left recursion was preferred to right recursion (contrary to recursive descent parsers, which are unable to process left recursion directly at all). The common good practice is modern grammarware engineering is to use iteration metalanguage constructs, sometimes referred to as EBNF iteration, since introduction of closures x^+ and x^* was among the differences between the original BNF and Extended BNF [Zay12a]. The use of metalanguage constructs is technology-agnostic, and the compiler compiler can make its own decisions about the particular way of implementation, and will neither crash nor have to perform any transformations behind the scenes. However, many existing grammars [Zay14a] contain yaccified definitions, and usually the first step in any project that attempts to reuse such grammars for practical purposes, starts with deyaccification.

distribute $\vdash$ ***DistributeAll:*** globally distribute sequential composition over choices. Perform **distribute**(x) for all $x \in \mathbf{N}$, such that $x ::= e_1 \in \mathbf{P}$, where $\exists e_2 = y_1 | \cdots | y_k$, such that $e_2 \neq e_1$ and $e_1 \approx e_2$ modulo factoring. The distribution operator aggressively pushes choices outwards, and is mostly used as a post-processing step after some other transformation that could introduce inner choices. Grouping such steps together and asserting the absence of local choices throughout the grammar in one sweep would be a different acceptable style of grammar manipulation.

eliminate $\vdash$ ***EliminateTop:*** eliminate unreachable nonterminal symbols. Perform **eliminate**(x) for all $x \in \mathbf{N}$, such that no derivations reaches x from the root. A corresponding transformation generator was proposed in [Zay10, §4.9] and [Zay11, §5.4]. A *top nonterminal* is a nonterminal that is defined but never used, and analysing and either connecting or removing top nonterminals from a grammar is a known technique in grammar recovery [LV01, SV00]. Especially in the situations when the root is known with certainty, we can assume all other tops to be useless, since they are unreachable from the starting symbol and are thus not a proper part of the grammar.

equate $\vdash$ ***EquateAll:*** merge all identically defined nonterminals. Perform **equate**(x, y) for all $x \in \mathbf{N}, y \in \mathbf{N}$, such that $\mathbf{P}_x \approx \mathbf{P}_y$, if x is indistinguishable from y. No explicit counterpart in prior work, but indeed the intent of using the **equate** operator is to ensure the lack of identical nonterminals in the grammar, which is effectively covered by this mutation.

fold $\vdash$ ***FoldMax:*** fold all foldable nonterminals. Perform **fold**(x) for all $x \in \mathbf{N}$, such that $\mathbf{P}_x = \{x ::= e\}$ and e occurs somewhere else in the grammar. No counterpart in prior work. A dangerous mutation that only makes sense in big grammars with complicated right hand sides of all production rules, otherwise it will result in too many unintended foldings.

horizontal $\vdash$ ***HorizontalAll:*** make all definitions horizontal. Perform **horizontal**(x) for all $x \in \mathbf{N}$, for which $|\mathbf{P}_x| > 1$. Discussed as a transformation generator in [Zay10, §4.9] and [Zay11, §5.4]. Grammar engineering often differentiates between *horizontal* [Zay10] or *flat* [LW01] definitions that consist of one production rule with a top level choice, and *vertical* or *non-flat* definitions that span over multiple production rules. Converting all production rules in $\mathbf{P}$ to

one form or the other to satisfy different grammarware frameworks preferences, is a commonly performed normalisation.

inline $\vdash$ ***InlineMax:*** inline non-recursive nonterminals. Perform **inline**(x) for all $x \in \mathbf{N}$ that have horizontal non-recursive definitions. All mutations that we considered above, were useful normalisations for refinement and improvement activities (sometimes referred to as "grammar beautification" [Zay10, Zay12b]). As we see now, this is not the only possible use for grammar mutations. Maximal unfolding of all nonterminal symbols is more than likely to make a grammar less readable, since it leads to overly bulky right hand sides. However, the minimal number of nonterminals for a given grammar, is a fundamental concept, since it denotes the limits of folding/unfolding [Zhu94], and it is a useful notion for grammar based metrics [ČKM+10], possibly related to the notion of grammatical levels [Gru71]. For smaller languages, as many DSLs are, it should be possible to inline all nonterminals except the root, converting a grammar to a form where $|\mathbf{N}| = 1$, thus allowing applications of theorems and techniques from the formal language theory, available only for one-nonterminal grammars, such as [JO09].

unchain $\vdash$ ***UnchainAll:*** get rid of all chain production rules. Perform **unchain**(p) for all $p \in \mathbf{P}$, such that $p = x ::= y$), where $x \in \mathbf{N}, y \in \mathbf{N}$ and $|\mathbf{P}_y| = 1$. Since unchaining is a form of limited inlining, its generalisation is an intentionally limited form of aggressive inlining. Reducing the number of chain production rules is one of the previously known ways of increasing readability of a grammar [LZ11].

unfold $\vdash$ ***UnfoldMax:*** unfold all non-recursive nonterminals. Perform **unfold**(x) for all $x \in \mathbf{N}$ that have horizontal non-recursive definitions. The same effect as for ***InlineMax*** above, but keeping the unfolded nonterminal symbols in the grammar makes undoing this mutation easier.

vertical $\vdash$ ***VerticalAll:*** make all definitions vertical. Perform **vertical**(x) for all $x \in \mathbf{N}$, for which $|\mathbf{P}_x| = 1$ and $\mathbf{P}_x = \{x ::= y_1 | \cdots | y_k\}$. A mutation with the opposite intent of ***HorizontalAll*** discussed above. A transformation generator with the same effect was previously proposed in [Zay10, §4.9] and [Zay11, §5.4].

3.2 Mutations of Type II: automated generalisation

The transformation operators **abstractize**, **appear**, **chain**, **concatT**, **concretize**, **define**, **inline**, **lassoc**, **rassoc**, **reroot**, **splitT** and **undefine**, when generalised to a mutation, require additional information, which can still be obtained automatically by additional traversals over the grammar, by pattern matching, etc. For the sake of simplicity, we define the Type II mutations with production rules as arguments, but they are trivially defined with any other kind of argument.

Definition 2 (auto-generalised grammar mutation) Given an operator $\tau = \langle \pi_p, \varphi_p \rangle$ parametrised with a production rule p, an auto-generalised mutation based on that operator, has the form $\mu_{\text{II}} = \langle \{\xi(p) \wedge \pi_{\psi(p)}\}, \{\varphi_{\psi(p)}\} \rangle$, where ξ is an additional constraint filtering out impossible arguments, and ψ is a rewriting rule preparing them to assume the form expected by the original precondition. ■

abstractize $\vdash$ ***RetireTs:*** remove all terminal symbols. For each $p \in \mathbf{P}$, traverse its right hand side for terminal symbols. If found, mark all of them and run **abstractize**(p'), where p' is the marked production rule. A corresponding transformation generator was used in [LZ09, §5.3],

[Zay10, §4.9, §4.10.6.1] and [Zay11, §5.4], as well as in one of the steps toward Abstract Normal Form [Zay12c, §3.1]. The resulting grammar mutation is particularly helpful when converging a concrete syntax and an abstract syntax of the same software language. While the abstract syntax definition may have differently ordered parameters of some of its constructs, and full convergence will require dealing them them and rearranging the structure with (algebraic) semantic-preserving transformations, we will certainly not encounter any terminal symbols and can thus safely employ this mutation.

appear $\vdash$ ***InsertLayout:*** make layout explicit. In parsing, it is common to perform layout-agnostic grammar manipulation and assume that the framework will insert optional layout preterminals between any two adjacent symbols in a context-free production rule. For layout-sensitive manipulation, it may be sensible to encode this step explicitly: that way, for instance, some other optimisation steps may take place before the grammar is fed to the parser generator.

chain $\vdash$ ***ChainMixed:*** separate chain production rules from unchained ones. For each $x \in \mathbf{N}$, if $|\mathbf{P}_x| > 1$, then for each non-chain production rule, perform **chain** on it. This is an important normalisation step for Abstract Normal Form [Zay12c, §3.1], which prepares the grammars for automated convergence. Basically, the guided convergence algorithm expects every nonterminal symbol to be either defined with one production rule, or with several chain production rules: this effect can be achieved with a mutation that generalises the **chain** operator.

concatT $\vdash$ ***ConcatAllT:*** concatenate all adjacent terminals. If there are any terminal symbols following one another sequentially, concatenate them. Terminals should be either all non-alphanumeric (the mutation would then mean the change of style in the language) or all alphanumeric (in which case a reasonable use case is grammar recovery [LV01, LZ11]).

concretize $\vdash$ ***ParenthesizeAll:*** introduce default concrete syntax. While the problem of deriving the abstract syntax from the concrete syntax is relatively well studied [Wil97], the reverse problem is far more complicated, since one abstract structure can be expressed with different concrete syntaxes. Without diving too much into the details of this issue, we can say that it is possible to automatically derive some kind of concrete representation by committing to homoiconicity and relying on M-expressions, S-expressions, Sweet-expressions, I-expressions, G-expressions and the like.

define $\vdash$ ***DefineMin:*** provide all missing definitions. For each $x \in \mathbf{N}$, if $\nexists p \in \mathbf{P}$ such that $p = x ::= e$, then define it as $x ::= \varphi$ (a metasymbol for failure, empty language, parse error). Since φ is a default semantics for any unknown nonterminal, since unknown nonterminals can be neither generated nor analysed, this mutation merely completes the grammar.

inline $\vdash$ ***InlineLazy:*** inline lazy (used once) nonterminals. One of the Type I mutations was ***InlineMax***: if we add an extra constraint to it that will make it applicable only to nonterminal symbols that are used just once throughout the whole grammar, we will get a very useful normalisation that can improve readability of automatically generated grammars and reduce the impact that the language documentation often has on a grammar (excessive number of secondary nonterminals introduced in order to spread the material better over the document sections). In particular, a transformation generator with a similar effect was proposed in [Zay10, §4.9] and [Zay11, §5.4].

lassoc $\vdash$ ***LAssocAll***, **rassoc** $\vdash$ ***RAssocAll:*** replace all iterations by associative equivalent. A simple iterative production rule has the form $x ::= x^+$, while its associative counterpart looks like $x ::= xx$, which can be complicated with expression labels and separators. Grammars influenced

by top-down parsing technology, such as definite clause grammars [PW80], often use explicit iteration constructs, which need to account for associativity when being converged [LZ09] with differently styled definitions of the same language. The difference between **lassoc** and **rassoc** is unnoticeable on the grammar level and is only relevant for coupled transformation of parse trees.

reroot $\vdash$ ***Reroot2top:*** reroot to top symbols. In the previous section, we have introduced the ***EliminateTop*** mutation that calculated the set of top (defined yet unused) symbols and eliminated them. An alternative way to treat top symbols is to make them new roots. A variation of this mutation is used for Abstract Normal Form in [Zay12c] with an additional requirement that a top nonterminal must not be a leaf in the relation graph. This is a rational constraint since a leaf top nonterminal defines a separated component.

splitT $\vdash$ ***SplitAllT:*** split all non-alphanumeric terminals. Any nonterminals that are either non-alphanumeric or mixed, can be split into sequences of terminals with their components. Intentional splitting and concatenating terminal symbols is common in automated grammar recovery [LV01, LZ11].

undefine $\vdash$ ***UndefineTrivial:*** remove trivially defined nonterminals. For each $x \in \mathbf{N}$, if $\forall p \in \mathbf{P}_x$, if $p = x ::= \alpha$ or $p = x ::= \varphi$ or $p = x ::= \varepsilon$, then **undefine**(x). This mutation has been proposed earlier as a step in normalisation to ANF [Zay12c, §3.1].

3.3 Mutations of Type III: narrowed generalisation

One of the issues with Type II mutations is that one operator generalises to one mutation, which is insufficient for operators that cover many similar cases. Let us recall that in [LZ09, LZ11] many operators were supported by an underlying equivalence relationship. For example, the class of *factor-equivalence* includes all differently factored but otherwise equal expressions; *massage-equivalence* is based on formulae like $xx^* \approx x^+$ and $x? \approx x|\varepsilon$. However, equivalence is often a symmetric relation, and only by looking at transformation arguments the direction of computation could be established. For grammar mutations, we need to virtually generate those arguments. Hence, it is impossible to define something like ***MassageAll***, because it would try to apply many patterns and rewrite them in both directions, which is obviously nondeterministic, not terminating and even pointless. Instead, we narrow the patterns used in the equivalence relation, fix the direction and define one mutation for each of those. The transformation operators that benefit from such processing, are: **iterate**, **factor**, **massage**, **narrow**, **permute**, **renameN**, **renameT** (the last two collectively called **renameX** for brevity), **widen** and **yaccify**. For some of them (most notably for **factor** and **renameX**, but also for **permute**) the shown narrowed cases are not exhaustive.

Definition 3 (narrowed grammar mutation) Given an operator $\tau = \langle \pi_p, \varphi_p \rangle$ parametrised with a production rule p, where φ is a composition of multiple rewritings $\varphi^{(i)}$, narrowed mutations based on that operator, have the form $\mu^{(i)}_{\mathrm{III}} = \langle \{\xi^{(i)}(p) \wedge \pi_{\psi^{(i)}(p)}\}, \{\varphi^{(i)}_{\psi^{(i)}(p)}\} \rangle$. Each $\xi^{(i)}$ is an additional constraint filtering out impossible arguments, and $\psi^{(i)}$ is a rewriting rule preparing them to assume the form expected by the original precondition. ■

iterate $\vdash \cdots$ (3 mutations). The **iterate** operator is a reverse of **lassoc/rassoc** that we have discussed in the previous section. The operator yields three narrowed mutations, since the un-

derlying equivalence relation is not symmetric and allows for three rewritings: $x ::= xyx \rightarrowtail x ::= (xy)^*x$, as well as $x ::= xyx \rightarrowtail x ::= x(yx)^*$ and $x ::= xx \rightarrowtail x ::= x^+$.

factor $\vdash$ ***distribute:*** automatic factoring outwards. For each $p \in \mathbf{P}$ such that $p = x ::= e$, where e contains inner choices and can be factored to the form $p' = x ::= y_1|\cdots|y_k$, then perform **factor**$(e, y_1|\cdots|y_k)$. This strategy is already implemented and resides in the original XBGF under the name **distribute**. Seeing how perfectly it fits the definition of a mutation, we must conclude that introducing such an intricate rewriting as a transformation operator in [LZ09] was not a correct language design decision.

factor $\vdash$ ***Undistribute:*** automatic factoring inwards. For each $p \in \mathbf{P}$, such that $p = x ::= y_1|\cdots|y_n$ and y_i are sequences that all share either starting or trailing nonempty elements such that $y_i = s_1 \cdots s_j y'_i t_1 \cdots t_k$, then perform **factor**$(y_1|\cdots|y_n, s_1 \cdots s_j(y'_1|\cdots|y'_n)t_1 \cdots t_k)$. The existence of a bidirectional but not bijective reverse of **distribute** only strengthens the point of view that treats it as a mutation, not as a transformation operator.

massage $\vdash \cdots$ (2×29 mutations). The **massage** operator is used for small-scale grammar refactorings. The massage-equivalence relation is symmetric and consists of algebraic laws not present in the trivial normalisations, such as $(x^+)? = x^*$ or $x^*x = x^+$ or $x? = x|\varepsilon$. These laws are defined declaratively, without any preference for the direction of computation, but by singling the laws out and fixing the direction, we get a family of twice the 29 mutations.

narrow $\vdash \cdots$ (5 mutations). This language-decreasing transformation operator rewrites $x?$ to x, x^+ to x, etc, for a total of five cases. An example of the resulting five narrowed mutations is ***NarrowStar2Opt*** that traverses the grammar in search for x^* and rewrites each of them to $x?$.

permute $\vdash \cdots$ (6 mutations). Dealing with different permutations is a common activity in converging grammars of various abstract syntaxes [LZ09] or in syntax directed translation with synchronous grammars [AU69]. In natural language processing permutations can be quite complicated and require discontinuous constituents (up to the point of enforcing the use of tree-adjoining grammars), but in software languages most permutations concern the operators being infix, prefix or postfix. This gives us six narrowed mutations. Their application is most useful in transcompiling languages with predominantly one kind of notation. Examples include translating from Forth to Infix Forth[4] by Andrew Haley (***PermutePostfix2Infix***) or from REBOL to Boron[5] by Karl Robillard (***PermuteInfix2Prefix***) and other kinds of *A-C unparsing* [BZ14], where one abstract syntax of a software language can be linked to several different concrete syntaxes.

renameX $\vdash \cdots$: naming conventions ($2 \times 6 \times 5$ mutations). Systematic renaming of nonterminals symbols has been discussed before in the context of coupled grammar mutations [Zay12b, §3.4] and transformation generators [Zay10, §4.9, §4.10.6.1] [Zay11, §5.4]. The same logic and the same conventions that apply to **renameN**, are also applicable to **renameT**. There are several different well-defined naming conventions for nonterminal symbols in current practice of grammarware engineering, in particular concerning multiword names. Enforcing a particular naming convention such as making all nonterminal names uppercase or turning camelcased names into dash-separated lowercase names, can be specified as a unidirectional grammar mutation (one for each convention). Every convention is specified by a word separator (almost exclusively space, dash, underscore, dot, slash or nothing) and a capitalisation policy (Capitalcase, lowercase, UP-

[4] http://www.webcitation.org/6GMSwBlr4
[5] http://urlan.sourceforge.net/boron/

PERCASE, CamelCase and mixedCase). These together give us 30 narrowed mutations per each **renameX** operator.

widen $\vdash \cdots$ (5 mutations). The reverse operator for **narrow** spawns five mutations that are bidirectional counterparts of the five discussed above.

yaccify $\vdash \cdots$ (2 mutations). While it was possible to implement ***DeyaccifyAll*** as a trivially generalised mutation, yaccification (replacing repetition with recursion) has two distinct rewriting strategies, favouring left and right recursion, respectively. Thus, we have two narrowed mutations: ***YaccifyAllL*** and ***YaccifyAllR***. This perfectly demonstrates the need for Type III mutations: **YaccifyAll** would work as a Type II mutation, but its behaviour would be nondeterministic. In the presence of parsing technologies with strong preference of left or right recursing, such a mutation would have been practically useless.

3.4 Mutations of Type IV: parametric generalisation

The last productive group of transformation operators includes those that cannot be generalised at all without additional information — which can still be provided by parametrising them. It is worth noting that some of the Type III mutations become more convenient to use when moved to Type IV. However, since it will require us to introduce more data types for representing operator notation for ***Permute***(X,Y) and ***RenameAllX***(nc_1,nc_2), we keep it simple, straightforward and automated. Further refinement of the implementation is always possible. This way, we also avoid the discussion of whether some operators should not be joined by introducing a discriminating argument — keeping the argument types confined to the types already found in the grammar representation of GrammarLab was a conscious language design decision.

Definition 4 (parametric grammar mutation) Given an operator $\tau = \langle \pi_p, \varphi_p \rangle$ parametrised with a production rule p, an parametrically generalised mutation based on that operator, has the form $\mu_{\text{IV}} = \langle \{\xi(a,p) \wedge \pi_{\psi(a,p)}\}, \{\varphi_{\psi(a,p)}\} \rangle$, where ξ is an additional constraint filtering out impossible arguments, ψ is a rewriting rule preparing them to assume the form expected by the original precondition, and a is an additional parameter. ■

The definition is trivially extended to cover other parameter types and different number of mutation parameters.

define $\vdash$ ***DefineAll***$([p_i])$. Similar to top nonterminals that we have reintroduced in Subsection 3.1, we can speak of *bottom* nonterminals [LV01, SV00], which are used but not defined in the grammar. This mutation effectively implements grammar composition, where one grammar defines the nonterminal needed by the other one.

disappear $\vdash$ ***DisappearEverywhere***(e). The **disappear** transformation operator removes an nillable ($x?$ or x^*) symbol from the given production rule. A somewhat more automated version of it would rewrite the whole grammar by making a given nillable symbol disappear from all production rules. In practice this step can be used to express the difference between the grammar of a software language used for parsing and another grammar used for pretty-printing [BV96].

eliminate $\vdash$ ***SubGrammar***$([x_i])$. Similar to program slicing [Wei81], one can perform *grammar slicing*. One of the most useful intentional slicing of a grammar is the one where the slice includes one of the nonterminals as the new root, and the transitively closed set of all nontermi-

nals reachable from it, with their corresponding production rules. Constructing a subgrammar starting with the already known roots is equivalent to ***EliminateTop***.

introduce $\vdash$ **importG**($[p_i]$). Introducing multiple fresh nonterminals in bulk would have been a generalisation of the **introduce** operator, but such a generalisation is already present in the original language under the name of **importG**. Just like **distribute** before, this begs for reconsideration of the appropriateness of treating **importG** as a transformation operator and not as a mutation.

redefine $\vdash$ ***RedefineAll***($[p_i]$). Just like **redefine** comprises the double effect of **undefine** and **define**, we can infer a mutation similar to **DefineAll**, but the one that disregards any existing conflicting definitions for the concerned nonterminals. Even though redefining a nonterminal has been studied in various frameworks before [DCMS02, KLV02, LZ11], the kind of grammar composition that **RedefineAll** offers, has never been explicitly considered.

undefine $\vdash$ ***SubtractG***($[p_i]$). Similarly, we can have a mutation that "subtracts" one grammar from the other one by undefining all nonterminals that are defined in the subtrahend. This mutation can be a way to resolve the well-known problem of modular language frameworks that often do not provide means to "exclude" a module after including it.

unite $\vdash$ ***UniteBySuffix***(a). Nonterminal names often carry meaning that was introduced by the grammar designer. One of the common grammar reengineering activities is unification of several nonterminals by grouping them semantically (in this case, nominally): `function_name`, `variable_name`, `task_name`, `library_unit_name`, etc[6], can be all united into a `name`; or `MethodModifier`, `VariableModifier`, `FieldModifier`, etc[7], into a `Modifier`. Considerable relaxation can also be achieved intentionally by using this mutation to combine all statements or all declarations.

4 Other kinds of mutations

The following transformation operators from XBGF were not used to build mutations based on them:

replace: in a sense, **replace** $\vdash$ **replace**. This transformation operator was meant to be the last resort in grammar programming, when a brutal change needs to be applied to a grammar. Most safe refinements of brutal replacing are already made into other operators, and the basic rewriting strategy is not generalisable any further.

importG: while we have seen **importG** in Subsection 3.4 being essentially a mutation such that *introduce* $\vdash$ **importG**, it is still feasible to treat it like a transformation operator, which can be rewritten into an even more general mutation. One of the possibilities would be automated grammar composition, when the choice of which grammar is imported, is made based on the grammar at hand. Further, much more detailed, investigation of such generalisations of both **importG** and **DefineAll** is needed before the result can be brought out to public.

The **extract** operator is essentially a composition of **introduce** and **fold**. It is perfectly clear

[6] Example taken from the Ada grammar, extracted from ISO/IEC 8652/1995(E) [Zay14a]. A browsable version is available at http://slps.github.io/zoo/ada/lncs-2219.html.

[7] Example taken from the Java grammar, extracted from Java Language Specification [LZ11, Zay14a]. A browsable version is available at http://slps.github.io/zoo/java/java-5-jls-read.html.

that **extract** generalises into a Type III family of mutations. However, it is far beyond trivial to propose a sensible, strongly motivated, non ad hoc list of useful extracting mutations. For example, the algorithm of converting a context-free grammar to Chomsky normal form, is obviously one of such **extract**-based mutations. However, in natural linguistics Chomsky normal forms of ranks higher than two are also considered (CNF of rank 3 also allows for production rules of the form $a ::= bcd$, etc), so we need to either propose a mutation suitable for such normalisations as well, or provide a separate mutation for each rank.

addH, addV, removeH, removeV: the intent behind adding or removing a particular piece to/from a grammar, is obvious and therefore not intentionally generalisable. In a parallel project, where we have tried to apply mining techniques to the existing transformation scripts in order to infer a higher level grammar transformation language Extended XBGF [Zay12c, §3.2.4], adding and removing branches was only spotted in a couple of stable use patterns, that involved straightforward superposition with **chain** or **vertical** — hardly any ground for generalisation to a mutation. It is possible that some very peculiar add-based mutations exist, but so far we have neither encountered nor needed them.

downgrade, upgrade, detour, clone, splitN: in the experience gained by previous grammar convergence projects, we can say that these transformation operators are also highly specific and always used once in a very limited scope. Generalisation of them to mutations seems impossible at the moment.

inject, project: more accurate versions of injection/projection were used as Type II mutations based on **concretize** and **abstractize**, as well as Type II and IV mutations based on **appear** and **disappear**. Further generalisation of injection/projection is possible, but seems too idiosyncratic to include here.

More complex mutations can be composed out of the already presented ones. Two basic composition schemes is possible. First, we could operate with triggers and rewriting rules directly and combine mutations in such a way that

$$\mu_1 \oplus \mu_2 = \langle \{\pi_i^{(1)}\}, \{\varphi_i^{(1)}\} \rangle \oplus \langle \{\pi_j^{(2)}\}, \{\varphi_j^{(2)}\} \rangle = \langle \{\pi_i^{(1)}\} \cup \{\pi_j^{(2)}\}, \{\varphi_i^{(1)}\} \cup \{\varphi_j^{(2)}\} \rangle.$$

Alternatively, we could just apply one mutation to the result of the other one and get $\mu_1 \circ \mu_2$. This always guarantees termination and compatibility.

The first way could be a more efficient manner in which we can present overly scattered Type III mutations. For example, for **massage** we could define a family of grouped narrowed mutations with ***MassageOpt*** for $x|\varepsilon \rightarrowtail x?$ and $x?|\varepsilon \rightarrowtail x?$ and $x?? \rightarrowtail x?$; ***MassageSepListLeft*** for $x(yx)^+ \rightarrowtail (xy)^+x$ and $x(yx)^* \rightarrowtail (xy)^*x$ ***MassageSepListRight*** for $(xy)^+x \rightarrowtail x(yx)^+$ and $(xy)^*x \rightarrowtail x(yx)^*$ etc. Then, essentially,

MassageOpt $=$ ***MassageChoice2Opt*** $\oplus$ ***MassageChoiceOpt2Opt*** $\oplus$ ***MassageOptOpt2Opt***.

The second way of composition is preferred for specifying mutations with multiple steps, consisting of multiple stages of normalisation. For instance, the Abstract Normal Form [Zay12c, §3.1] needed for guided grammar convergence [Zay14b], would look like this:

$$\begin{aligned} \textit{ANF} = {} & \textit{RetireTs} \circ \textit{DistributeAll} \circ \textit{VerticalAll} \circ \textit{UndefineTrivial} \circ \textit{ChainMixed} \\ & \circ \textit{MassageSepListPlus2L} \circ \textit{MassageSepListStar2L} \circ \textit{Reroot2top} \circ \textit{EliminateTop} \end{aligned}$$

If this seems complicated, remember that each of these steps is an intentional rewriting on its own, so each component of this definition maps directly to a mutation and to an objective

that needs to be satisfied by the normal form. For the purposes of guided grammar convergence [Zay14b], this normalisation needs to be implemented with origin tracking mechanisms to preserve the correspondence between the nonterminals in the normalised and in the reference grammar. Still, the formula above remains the definition, and the implementation generates bidirectional grammar transformation steps [Zay12b].

5 Dark data: a non-intentional approach

Dark data is information about failed scientific experiments [Goe07]. In this section, we will briefly describe a previously failed attempt to design a new software language for grammar manipulation with maintainability in mind. This attempt has costed several months of work, but has only been documented before in a technical report [Zay12c, §3.2.4].

In [LZ11], we have undertaken the biggest semi-automated grammar transformation project known up to date. It consists of over 1600 grammar transformation operator calls, used to converge six standardised grammars of different versions of the Java programming language. Such scale raises a question of maintaining these transformation chains. We had to solve these question ourselves during rethinking of the underlying approach when we upgraded the first version of the project report into a bigger journal paper that required substantially deeper insights and more universal applicability.

The first attempt to design a new language was based on analysing patterns that occurred in those transformation scripts and defining new operators for them as (meta)syntactic sugar — in fact, it was a vertical DSL that gave concise notation to lengthier combinations of low-level transformation steps. The idea was that having low-level operators such as **unfold**, **introduce** or **factor** is useful for understanding the underlying semantics and sharing small illustrative examples, but higher level operators are needed for advanced maintenance and evolution activities performed on the transformation sequences themselves. For example, consider this scenario:

```
A ::= B C D E ;              A ::= B C DF E ;
A ::= B C F E ;      ⟹      A ::= G ;
A ::= G ;                    DF ::= D ;
                             DF ::= F ;
```

Pulling out such `DF` requires factoring the defining expressions of `A` and then extracting `DF`, but since factoring is only defined on choices, we would also need to horizontalise `A` before and verticalise `DF` after the extraction. All these steps can be combined in one parametrically defined transformation operator which we may call **exbgf:pull-out**. In the Java Language Specification convergence scenario [LZ11] we have identified 23 occurrences of possible use for **exbgf:pull-out**, each one replacing three (**factor** ∘ **extract** ∘ **vertical**) or five (**horizontal** ∘ **factor** ∘ **vertical** ∘ **extract** ∘ **vertical**) XBGF transformation steps.

As a result, the line count and statement (function application) count went down considerably, as one can see on Table 1: introducing many high level operators like **exbgf:pull-out**, has reduced the number of atomic transformation steps that need to be defined by a grammar engineer in order to achieve convergence, by 25% on average. EXBGF (Extended XBGF) was implemented purely as a vertical DSL, which means that from the EXBGF script programmed by a grammar engineer, an equivalent longer XBGF script is generated automatically and subsequently executed. Such generated XBGF also turned out to be not much less efficient than the

	jls1	jls2	jls3	jls12	jls123	r12	r123	Total
XBGF, LOC	682	6774	10721	5114	2847	1639	3082	30859
EXBGF, LOC	399	5509	7524	3835	2532	1195	2750	23744
	−42%	−19%	−30%	−25%	−11%	−27%	−11%	−23%
genXBGF, LOC	516	5851	9317	4548	2596	1331	2667	26826
	−24%	−14%	−13%	−11%	−9%	−19%	−13%	−13%
XBGF, nodes	309	3,433	5,478	2,699	1,540	786	1,606	15851
EXBGF, nodes	177	2,726	3,648	1,962	1,377	558	1,446	11894
	−43%	−21%	−33%	−27%	−11%	−29%	−10%	−25%
genXBGF, nodes	326	3,502	5,576	2,726	1,542	798	1,610	16080
	+6%	+2%	+2%	+1%	+0.1%	+2%	+0.3%	+1%
XBGF, steps	67	387	544	290	111	77	135	1611
EXBGF, steps	42	275	398	214	98	50	120	1197
...pure EXBGF	27	104	162	80	30	34	44	
...just XBGF	15	171	236	134	68	16	76	
	−37%	−29%	−27%	−26%	−12%	−35%	−11%	−26%
genXBGF, steps	73	390	555	296	112	83	139	1648
	+9%	+1%	+2%	+2%	+1%	+8%	+2%	+2%

Table 1: Size measurements of the Java grammar convergence case study, done in XBGF and in EXBGF. In the table, XBGF refers to the original transformation scripts, EXBGF to the transformations in Extended XBGF, genXBGF measures XBGF scripts generated from EXBGF. LOC means lines of code, calculated with `wc -l` on pretty-printed XML; nodes represent the number of nodes in the XML tree, calculated by XPath; steps are nodes that correspond to transformation operators and not to their arguments. Percentages are calculated against the XBGF scripts of the original study.

manually coded XBGF: the difference was within 4%.

Unfortunately, the claims about boosting readability and comprehensibility of transformation scripts, were never realised. The main problems were:

- the new operators were not orthogonal (patterns tended to overlap) and did not form a uniform system; this led to them having less predictable behaviour, being not trivial to learn and impossible to deterministically automate migration of low-level scripts to the new language;

- many XBGF operators were already semantically refined versions of their more brutal counterparts (e.g., folding is a very special case of replacement), and from the language engineering point of view, it was unclear where to make the cut: for instance, should **redefine** be a part of XBGF and pose additional semantic constraints or should it be a part of EXBGF and be a straightforward superposition of **undefine** and **define**?;

- EXBGF operators were defined statically — only the instantiation of the EXBGF operator could see the input grammar, while the semantics of the EXBGF operator itself has to be defined only in terms of its parameters; the consequence was that operators such as **distribute** that had a high- level feel, were impossible to specify as EXBGF operators;

Beside all of the above reasons, we faced a serious threat to validity since the new language was basically a result of applying clone detection techniques to one case study, and there was no other case study of a comparable size to validate and calibrate the findings.

The interested readers are redirected to the open source repository of the Software Language Processing Suite [ZLS+14] where all the necessary data is already deposited as http://github.com/grammarware/slps/tree/master/topics/convergence/java.

6 Conclusions

If any programmable grammar transformation is an application of an appropriately parametrised transformation operator, then a grammar mutation is an application of a well-understood rewriting algorithm which can be automated: systematically renaming all nonterminals or fetching a subgrammar can serve as examples of grammar mutations. Effectively, the whole input grammar becomes a parameter. Grammar mutations can be used to improve automation of transformation scenarios by generating a part of the required transformation chain automatically, and they are also useful in normalisations.

In this paper, we have analysed XBGF [LZ09, LZ11], a language consisting of **55** transformation operators, in the spirit of intentional software [SCC06], and classified **12** operators as Type I, **15** operators as Type II, **12** operators as Type III and **7** operators as Type IV. In this process **36** operators were used once, **5** were used twice in different contexts and **14** were left unused and claimed nongeneralisable. Since every operator of Types I, II and IV gives us one mutation, but each Type III operator spawns 2–58 narrowed mutations, we ended up inferring **235** elementary grammar mutations. All of them were briefly introduced on the pages of the paper with claims concerning their use case scenarios in previously published work by grammar engineers. Yet only **42** of them have been explicitly studied before. We have identified two operators (**importG** and **distribute**) that are, in fact, grammar mutations, which partially explains why they resisted straightforward bidirectionalisation in [Zay12b]. The remaining **191** mutations are ready to be used and to be investigated further. The rules for composing elementary mutations into more complicated normalisations, were also included in Section 4.

The intentional rewriting schemes presented here, are not limited to bare formalisations. Metaprogramming techniques helped us to infer the implementations of all mutations from the implementations of corresponding operators, in an automated fashion with the unavoidable gaps (the ξ and ψ in the formulae) programmed manually. Nevertheless, it took considerably less effort than usually expected under such circumstances, and provided stronger guarantees about the completeness and orthogonality of the designed language. These guarantees rely on the source language, which in our case survived prior validation by large case studies; by comparison to previously existing languages in the same domain; by bidirectionalisation; and by building a negotiated framework on top of it [LZ11, Zay12b, Zay14c].

Currently the obtained set of grammar mutations is being actively used in maintenance and reengineering of grammars from the Grammar Zoo, a repository containing 550+ syntax definitions, AST specifications, metamodels, data schemata, etc. Further research on grammar mutation will include composition of practical complex mutations from the available elementary ones and possibly incorporating grammar mutations and grammar transformations alike in

one uniform grammar manipulation language. After intensive testing and documentation, both transformation and mutations suites will be added to Rascal language workbench [KSV11].

From this case study it is not yet clear whether the "⊢" operation on its own can be generalised as a higher order combinator. It is also not apparent whether the four types of grammar mutations that we have identified here, were feasible to infer automatically in some way, possibly by use of genetic algorithms. We reserve bigger issues like these to future work, as well as adjacent topics like "co-generalisation" of test sets.

The global concept of automated software language engineering also requires more scrupulous investigation. It remains to be seen whether any given software language can be generalised systematically in the same fashion. (Can we infer C from Assembly and C++ from C? Would inferred C++ as "C syntax with the OOP intention" be different from the real C++? In what way?). In particular, within this case study we have overcome the fundamental limitations of transformational design [Voe01] by committing to a semi-automated (and not fully automatic) process controlled by a human expert performing classification of operators and providing additional information. This case study demonstrates the viability of the method, but not its limits.

Bibliography

[AU69] A. V. Aho, J. D. Ullman. Syntax Directed Translations and the Pushdown Assembler. *Journal of Computer and System Sciences* 3(1):37–56, 1969.

[BV96] M. van den Brand, E. Visser. Generation of Formatters for Context-Free Languages. *ACM Transactions on Software Engineering Methodology (ToSEM)* 5(1):1–41, 1996.

[BZ14] A. H. Bagge, V. Zaytsev. A Bidirectional Megamodel of Parsing. 2014. Submitted to ECMFA 2014.

[ČKM+10] M. Črepinšek, T. Kosar, M. Mernik, J. Cervelle, R. Forax, G. Roussel. On Automata and Language Based Grammar Metrics. *Computer Science and Information Systems* 7(2):309–329, 2010.

[DCMS02] T. R. Dean, J. R. Cordy, A. J. Malton, K. A. Schneider. Grammar Programming in TXL. In *Proceedings of the Second International Workshop on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2002.

[Goe07] T. Goetz. Freeing the Dark Data of Failed Scientific Experiments. *Wired* 15(10), 2007.

[Gru71] J. Gruska. Complexity and Unambiguity of Context-free Grammars and Languages. *Information and Control* 18(5):502–519, 1971.

[Hoa73] C. A. R. Hoare. Hints on Programming Language Design. Technical report, Stanford University, Stanford, CA, USA, 1973.

[JM01] M. de Jonge, R. Monajemi. Cost-Effective Maintenance Tools for Proprietary Languages. In *Proceedings of 17th International Conference on Software Maintenance (ICSM 2001)*. Pp. 240–249. IEEE, 2001.

[JO09] A. Jeż, A. Okhotin. One-Nonterminal Conjunctive Grammars over a Unary Alphabet. In Frid et al. (eds.), *Computer Science — Theory and Applications*. LNCS 5675, pp. 191–202. Springer, 2009.

[Joh75] S. C. Johnson. *YACC—Yet Another Compiler Compiler*. Computer Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, New Jersey, 1975.

[KLV02] J. Kort, R. Lämmel, C. Verhoef. The Grammar Deployment Kit. In Brand and Lämmel (eds.), *Proceedings of LDTA 2001*. ENTCS 65. Elsevier, 2002.

[KLV05] P. Klint, R. Lämmel, C. Verhoef. Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering Methodology (ToSEM)* 14(3):331–380, 2005.

[KSV11] P. Klint, T. van der Storm, J. Vinju. EASY Meta-programming with Rascal. In Fernandes et al. (eds.), *Post-proceedings of GTTSE 2009*. LNCS 6491, pp. 222–289. Springer, Jan. 2011.

[Läm01a] R. Lämmel. Grammar Adaptation. In *International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*. LNCS 2021, pp. 550–570. Springer, 2001.

[Läm01b] R. Lämmel. Grammar Testing. In Hussmann (ed.), *Proceedings of the Conference on Fundamental Approaches to Software Engineering (FASE'01)*. LNCS 2029, pp. 201–216. Springer, 2001.

[LV01] R. Lämmel, C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience* 31(15):1395–1438, December 2001.

[LW01] R. Lämmel, G. Wachsmuth. Transformation of SDF Syntax Definitions in the ASF+SDF Meta-Environment. In *Proceedings of LDTA 2001*. ENTCS 44. Elsevier Science, 2001.

[LZ09] R. Lämmel, V. Zaytsev. An Introduction to Grammar Convergence. In Leuschel and Wehrheim (eds.), *Integrated Formal Methods (iFM 2009)*. LNCS 5423, pp. 246–260. Springer, Feb. 2009.

[LZ11] R. Lämmel, V. Zaytsev. Recovering Grammar Relationships for the Java Language Specification. *Software Quality Journal (SQJ)* 19(2):333–378, Mar. 2011.

[MHS05] M. Mernik, J. Heering, A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys* 37(4):316–344, 2005.

[PW80] F. C. N. Pereira, D. H. D. Warren. Definite Clause Grammars for Language Analysis. A Survey of the Formalism and a Comparison with Augmented Transition Networks. *AI* 13(3):231–278, 1980.

[SCC06] C. Simonyi, M. Christerson, S. Clifford. Intentional Software. In *Conference on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA '06, pp. 451–464. ACM, 2006.

[SV00] M. P. A. Sellink, C. Verhoef. Development, Assessment, and Reengineering of Language Descriptions. In Ebert and Verhoef (eds.), *Proceedings of CSMR*. Pp. 151–160. IEEE, Mar. 2000.

[VBD+13] M. Völter, S. Benz, C. Dietrich, B. E. M. Helander, L. C. L. Kats, E. Visser, G. Wachsmuth. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[Voe01] J. P. M. Voeten. On the Fundamental Limitations of Transformational Design. *ACM Transactions on Design Automation of Electronic Systems* 6(4):533–552, Oct. 2001.

[Wei81] M. Weiser. Program Slicing. In *Proceedings of the Fifth International Conference on Software Engineering (ICSE)*. Pp. 439–449. IEEE Press, San Diego, California, United States, Mar. 1981.

[vW65] A. van Wijngaarden. Orthogonal Design and Description of a Formal Language. MR 76, SMC, 1965.

[Wil97] D. S. Wile. Abstract Syntax from Concrete Syntax. In *Proceedings of ICSE*. Pp. 472–480. ACM, 1997.

[Wir74] N. Wirth. On the Design of Programming Languages. In *IFIP Congress*. Pp. 386–393. 1974.

[Zay10] V. Zaytsev. *Recovery, Convergence and Documentation of Languages*. PhD thesis, VU, 2010.

[Zay11] V. Zaytsev. Language Convergence Infrastructure. In Fernandes et al. (eds.), *Post-proceedings of GTTSE 2009*. LNCS 6491, pp. 481–497. Springer, Jan. 2011.

[Zay12a] V. Zaytsev. BNF WAS HERE: What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions. In Ossowski and Lecca (eds.), *Proceedings of SAC*. Pp. 1910–1915. ACM, 2012.

[Zay12b] V. Zaytsev. Language Evolution, Metasyntactically. *EC-EASST* 49, 2012.

[Zay12c] V. Zaytsev. The Grammar Hammer of 2012. *Computing Research Repository (CoRR)* 4446:1–32, 2012.

[Zay14a] V. Zaytsev. Grammar Zoo: A Repository of Experimental Grammarware. 2014. Submitted to the Fifth Special issue on Experimental Software and Toolkits of Science of Computer Programming (SCP EST5).

[Zay14b] V. Zaytsev. Guided Grammar Convergence. In *Journal of Object Technology*. 2014. In print.

[Zay14c] V. Zaytsev. Negotiated Grammar Evolution. *JOT Special Issue on Extreme Modeling*, 2014. In print.

[Zhu94] H. Zhu. How Powerful are Folding/Unfolding Transformations? *JFP* 4(01):89–112, 1994.

[ZLS+14] V. Zaytsev, R. Lämmel, T. van der Storm, L. Renggli, G. Wachsmuth. Software Language Processing Suite[8] 2008–2014. http://slps.github.io.

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