Electronic Communications of the EASST
Volume 73 (2016)

Graph Computation Models
Selected Revised Papers from GCM 2015

Cotransforming Grammars with Shared Packed Parse Forests

Vadim Zaytsev

21 pages

Guest Editors: Detlef Plump
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Cotransforming Grammars with Shared Packed Parse Forests

Vadim Zaytsev

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

Abstract: SPPF (shared packed parse forest) is the best known graph representation of a parse forest (family of related parse trees) used in parsing with ambiguous/conjunctive grammars. Systematic general purpose transformations of SPPFs have never been investigated and are considered to be an open problem in software language engineering. In this paper, we motivate the necessity of having a transformation operator suite for SPPFs and extend the state of the art grammar transformation operator suite to metamodel/model (grammar/graph) cotransformations.

Keywords: cotransformation, generalised parsing, parse graphs

1 Motivation

Classically, parsing consumes a string of characters or tokens, recognises its grammatical structure and produces a corresponding parse tree [ASU85]. A more modern perspective is that parsing recognises structure and expresses it explicitly [ZB14]. In many situations, trees appear to be unsatisfactory target data structures: they can express hierarchy easily, but any other structural commitments require special tricks and encodings, which are much less preferable than switching to graphs or pseudographs [SL13]. The most common scenarios include expressing uncertainty (e.g., in generalised parsing), maintaining several structural views (e.g., in the style of Boolean grammars) or manipulating recursive structures (e.g., with structured graphs).

Generalised parsing algorithms (GLR [Tom85], SGLR [Vis97], GLL [SJ10a], RIGLR [SJ05], etc.) differ from their classic counterparts in dealing with ambiguity [BSVV02, BV11]: instead of trying to avoid, ignore or report ambiguous cases, they are expressed explicitly in so called parse forests. Formally, a parse forest is a set of equally grammatically correct parse trees. Some of them may be semantically different, which makes such ambiguity significant and usually undesirable. In practice, such sets usually need to be filtered or ranked in order to make full use of the available tree-based approaches to program analysis and transformation. In *Boolean grammars* [Okh04] and *conjunctive grammars* [Okh01], we can use conjunctive clauses in a grammar to explicitly specify several syntactically different yet equally grammatical views of the same input fragment — they can be semantically equivalent [SC15] or one branch strictly more expressive than the other [Zay13]. Parsing techniques can utilise such specifications to create special kinds of nodes in a parse tree whose descendant subtrees share leaves [Okh13, Ste15]. Shared recursive structures are also facilitated by *parametric higher-order abstract syntax* [PE88, DPS97, Chl08]. It is an advanced method with high expressiveness, but it often requires similarly advanced techniques like multilevel metareasoning [MA03] and demands the use of automated theorem provers [DFH95, RHB01]. For now we will focus on the first two cases, since both kinds of structures defined by those two related approaches conceptually are *parse forests*.

This paper is an endeavour to advance the theory and practice of transformation of parse forests — specifically, coupled transformations (in short, cotransformations) of *grammars* as structural definitions of ambiguous languages and *graphs* representing sets of trees conforming to them. This domain of software language engineering is underdeveloped, which was pointed out as one of the major open problems in modern software language engineering by James Cordy and explained in his recent keynote at the OOPSLE workshop [BZ15, §3.1]. In the next section, we will provide some minimal background knowledge needed to appreciate the rest of the paper. Then, in Section 3 we will define the problem we can solve and its implications. Section 4 is the main section which introduces the operator suite for grammar-based gardening and proposes a classification of its operators. Section 5 contains links to related and future work, and Section 6 concludes the paper.

2 Background

There have been various attempts in the past to represent parse forests. The earliest ones required a grammar to be in a Chomsky Normal Form [Cho59] — theoretically a reasonable assumption since any context-free grammar can be normalised to CNF, but ultimately we need a parse forest for the original grammar, not for the normalised one, which would require bidirectional grammar transformations [Zay12] to be coupled with tree and forest transformations. Such a setup is far from trivial and thus not practically feasible.

The next attempt in representing parse forests revolved around tree merging [Ear68]: such a parse forest representation would result in a tree-like DAG with all the edges of all the trees in the forest. This is obviously an overapproximation of the forest (see Figure 1), which requires additional information in order to be unfolded into a set of trees — in other words, in order for any sensible manipulation to happen. Obviously, having a data structure that requires so much nontrivial postprocessing overhead, is highly undesirable.

The best representation of a conceptual parse forest (a set of trees with equal lists of leaves) so far is a so-called *shared packed parse forest* [Tom85, §2.4], SPPF from now on: its components are merged from the top until the divergent nodes, and due to maximal sharing the leaves and perhaps even entire subtrees grouping leaves together, are also merged. An example of such a graph is given on Figure 2. Formally, an SPPF is an acyclic ordered directed graph where each edge is a tuple from a vertex to a linearly ordered list of successors and each vertex may have more than one successor list. If V is a set of vertices, then edges are:

$$E = \{\langle v_i, (v_{i1}, v_{i2}, ..., v_{ik_i})\rangle \mid v_i \in V, v_{ij} \in V\} \subseteq V \times V^*$$

SPPF-like structures are used nowadays both in software language toolkits that allow explicit ambiguities, such as Rascal [KSV09], and those that allow explicit conjunctive clauses, such as TXL [Ste15]. For a detailed view on the implementation details we refer the readers to a paper on ATerms [BJKO00]. The theory of transformations of SPPFs is underdeveloped — they work exactly like transformations on trees if no shared nodes are present around the transformation point, and are unreliable and unpredictable otherwise. We would like to improve the situation by providing a toolkit for forest *migration* — i.e., transformations of SPPFs inferred from grammar evolution steps.

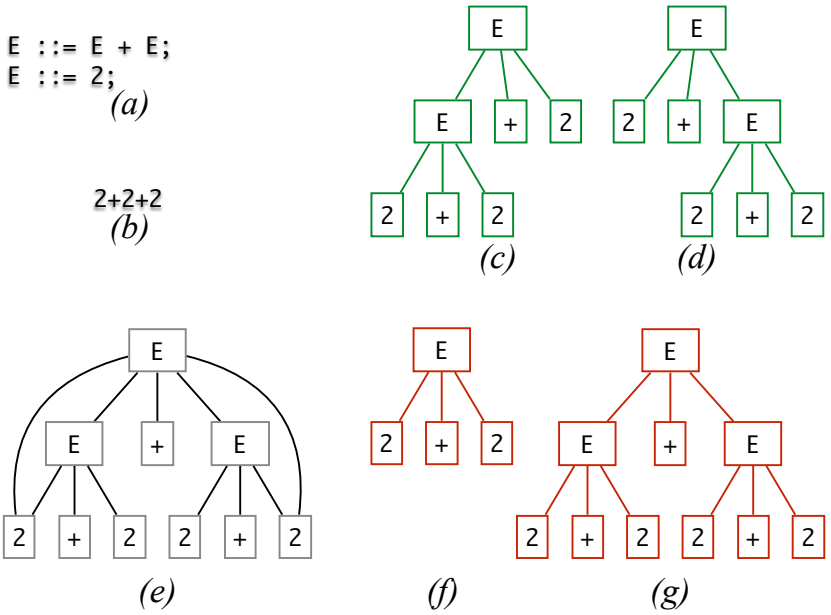
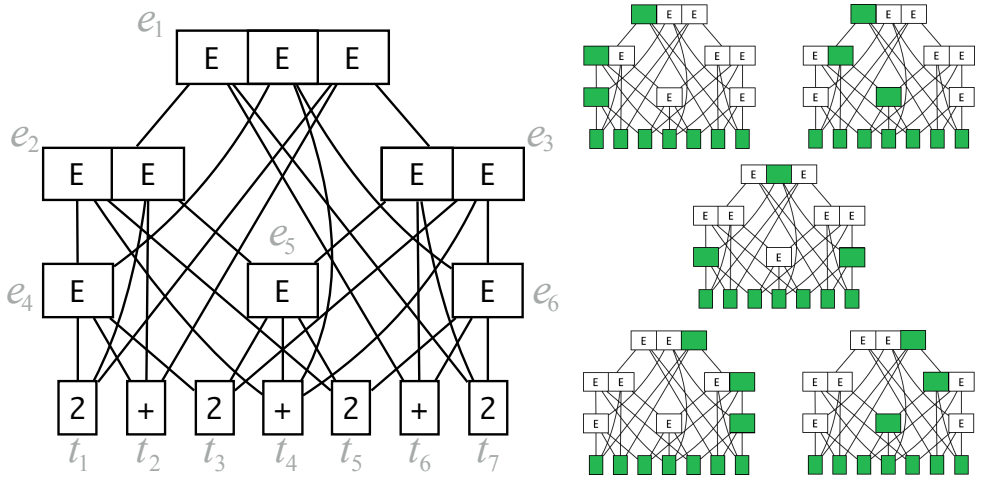


Figure 1: Demonstration that the Earley representation overapproximates parse forests: **(a)** a simple ambiguous grammar example; **(b)** a term with ambiguous parse; **(c)&(d)** correct parses; **(e)** the graph representation of the forest suggested by Earley [Ear68]; **(f)&(g)** incorrect parse trees that are well-formed according to the grammar (a) and covered by the parse tree representation (e), but not corresponding to the actual term (b).

3 Transformation

For many years trees have been the dominant data structure for representing hierarchical data in software language processing. They are remarkably easy to define, formalise, implement, validate, visualise and transform. There are many ways to circumvent data representation as graphs by considering a tree together with a complementary component such as a relation between its vertices that would have turned a tree into a cyclic graph, as well as many optimisations of graph algorithms that work on skeleton trees of a graph. Take, for instance, traversing a tree — it can be done hierarchically from the root towards the leaves or incrementally from the leaves towards the root, each case guaranteed termination even if the traversal is not supposed to stop when a match is made. This naturally provides us with four traversal strategies found in metaprogramming: bottom-up-continue, bottom-up-break, top-down-continue and top-down-break [VBT98, BHKO02, KSV09]. More sophisticated and flexible traversal strategies exist but are less in demand — for powerful strategic programming frameworks and paradigms we refer the readers to Stratego [Vis01], Strafunski [LV03], SYB [LJ04], Cω [BMS05], Nuthatch [BL13], structure-shy programs [CV11], generic FP [JJ97, Hin00], etc. A detailed and only slightly outdated overview of visiting functions, strategic programming and typed/untyped rewriting can be found in the work of van den Brand et al [BKV03] and the bibliography thereof. This section is focused on finding existing techniques that can be or are in fact SPPF transformations.



$$\{\ \langle e_1,(e_2,t_6,t_7)\rangle, \langle e_1,(e_4,t_4,e_6)\rangle, \langle e_1,(t_1,t_2,e_3)\rangle,$$
$$\langle e_2,(e_4,t_4,t_5)\rangle, \langle e_2,(t_1,t_2,e_5)\rangle, \langle e_3,(e_5,t_6,t_7)\rangle, \langle e_3,(t_3,t_4,e_6)\rangle,$$
$$\langle e_4,(t_1,t_2,t_3)\rangle, \langle e_5,(t_3,t_4,t_5)\rangle, \langle e_6,(t_5,t_6,t_7)\rangle\ \}$$

Figure 2: One the left, an SPPF graph resulted from parsing the input "2+2+2+2" with the grammar from Figure 1 (a). On the right, there are five parse trees in a forest, which are packed in a triple ambiguity, two of subgraphs of which have double ambiguities. All of them share leaves and subtrees whenever possible. Below the pictures we show its formal representation as an ordered directed graph.

3.1 Disambiguation

One of the relatively well-researched kind of SPPF transformations is disambiguation — it is commonly practised with ambiguous generalised parsing because static detection of ambiguity is undecidable for context-free grammars [Can62]. However, most of the time the intention of an average grammarware engineer is to produce one parse tree, so this line of research is mostly about leveraging additional sources of information to obtain a parse tree from a parse forest. There are three main classes of disambiguation techniques:

- Ordered choice, dynamic lookahead and other conventions aimed to *prevent* ambiguities altogether or avoid them. These are fairly static, relatively well-understood and widely used in TXL [DCMS02], ANTLR [PF11] and PEG [For04].
- Follow/precede restrictions, production rule priorities, associativity rules and other annotations for local sorting (preference, avoidance, priorities) that help to prune the parse forest *during* its creation. Since these are algorithmic approaches in a sense that they modify the generation process of an SPPF and thus are not proper mappings from SPPFs to SPPFs, we will not consider them in the rest of the paper and refer to other sources primarily dedicated to them [BSVV02, BV11].
- Disambiguation filters that are run *after* the parsing process has yielded a fully formed

SPPF: their main objective is to reduce the number of ambiguities and ultimately to shave all of them off, leaving one parse tree. An example of this would be how processing production rules marked for rejection is done for SGLR [BSVV02] and GLL [BV11] — even though recursive descent parsers can handle an equivalent construct (and-not clause) during parsing without any trouble [SC15].

Formally speaking, the first class never produces parse forests; the second class works with disambiguators (higher order functions that take a parser and return a parser that produces less ambiguous SPPFs) [BSVV02]; the third class uses filters (functions that take an SPPF and produce a less ambiguous SPPF) [KV94]. In some sources approaches with disambiguators are called "semantics-directed parsing" and approaches with filters are called "semantics-driven disambiguation" [BKMV03], since both indeed rely on semantic information to aid in the syntactic analysis. Disambiguation filters are still but a narrow case of SPPF transformation, but they have apparent practical application and are therefore well-researched.

3.2 Grammar programming

Grammar programming is like normal programming, but with grammars: there is a concrete problem at hand which can be solved with a grammar, which is then being adjusted until an acceptable solution emerges. A representative pattern here is working with a high level software artefact describing a language (we assume it to be a grammar for the sake of simplicity, but in a broad sense it can be a schema, a metamodel, an ontology, etc), from which a tool solving the problem at hand is inferred automatically.

There are at least three common approaches to grammar programming: manual, semi-automated and operator-based. *Manual grammar programming* involves textual/visual editing of the grammar file by a grammarware engineer. It is the easiest method in practice and is used quite often, especially for minor tweaks during grammar debugging. However, it leads to hidden inconsistencies within grammars (which require advanced methods like grammar convergence to uncover [LZ11]), between changed grammars and cached trees (which demand reparsing) and between grammars and program transformations (which requires more manual labour). *Semi-automated grammar programming* adds a level of automation to that and thus is typically used in scenarios when a baseline grammar needs to be adjusted in different ways to several tasks (parsing language dialects, performing transformations, collecting metrics, etc). Usually the grammarware toolkit provides means to extend the grammar or rewrite parts of it — examples include TXL [DCMS02], GDK [KLV02] and GRK [Lä05]. Arguably the latter two of these examples also venture into the next category since they contain other grammar manipulation instruments like folding/unfolding. If we extend this arsenal with even more means like merging nonterminals, removing grammar fragments, injecting/projecting symbols from production rules, chaining/unchaining productions, adding/removing disjunctive clauses, permuting the order and narrowing/widening repetitions, we end up having an *operator suite* for grammar programming. The advantage of having such a suite lies in the simple fact that each of the operators can be studied and implemented in isolation, and the actual process of grammar programming will involve calling these operators with proper arguments in the desired order. Examples of operator suites include FST [LW01], XBGF [LZ11], ΞBGF [Zay12] and SLEIR [Zay14b].

3.3 Cotransformation

We speak of cotransformations when two or more kinds of mutually dependent software artefacts are transformed together to preserve consistency among them: usually one changes, and others co-evolve with it [Läm04]. Naturally, the first cotransformation scenario we should think of, involves an SPPF and a grammar that defines its structure. This change can be initiated from either side, let us consider both.

Assuming that we have a sequence of grammar transformation steps, we may want to execute them on the language instances (programs) as well, to make them compatible with the updated grammar. Such a need arises in the case of grammar convergence [LZ09], when a relationship between two grammars is reverse engineered by programming the steps necessary to turn one into the other, and a co-transformation can help to migrate instances obtained with one grammar to fit with the other. For example, we could have a grammar for the concrete syntax and a schema for serialisation of the same data — a transformation sequence that strips the concrete grammar from elements not found in the schema (typically terminals guiding the parsing process such as semicolons and brackets), could also be coupled with a transformation sequence that removes the corresponding parts from the graphs defined by them (e.g., a parse tree and an XML document).

Consider another scenario where we have the change on language instances and want to lift it to the level of language definitions. An example could be found in program transformation, a common software engineering practice of metaprogramming. If we want a refactoring like extracting a method, renaming a variable or removing a go-to statement, it is easy and practical to express it in terms of matching/rewriting paradigm: in Spoofax [KV10], Rascal [KSV09], TXL [DCMS02], ATL [JAB+06], XSLT [Kay07], etc. However, a correct refactoring should preserve the meaning of the program, and the first step towards that is syntactic correctness of this program. For non-refactoring transformations found in aspect-oriented development, automated bug fixing and other areas, we still want to ascertain the extent to which the language is extended, reduced or revised. In the case of strongly typed metaprogramming languages, they will not allow you to create any ill-formed output, but the development process can lead you to *first* specify a breaking transform and *then* cotransform the grammar so that it "fits" — which is what cotransformations are good for.

3.4 Explicit versus implicit

This was already mentioned before, but becomes a crucial point from now on: parse forests can arise from two different sources — conjunctive clauses in the grammar used for parsing and generalised parsing with ambiguous grammars. The latter case can be considered implicit conjunction, since it is present on the level of language instances but not on the grammar level. In that case, instead of a more cumbersome construction specifying a precise parse, we use a simpler grammatical definition which yields a forest. If a grammar is both conjunctive and ambiguous, this can lead to its both implicit and explicit conjunctive clauses to be found in SPPFs — with no observable difference on an instance level.

Similarly, some of the transformations will "collapse" conjunctions, making one branch of a clause equal to another. Formally, for an SPPF node to have several branches means existence of several edges in the form $\langle v_i, (v_{i1}, \ldots, v_{ik_i})\rangle$, $\langle v_i, (v'_{i1}, \ldots, v'_{ik'_i})\rangle$, etc. When a transformation results

	Language preserved	Language extended	Language reduced	Language revised
SPPFs preserved	bypass eliminate introduce import vertical horizontal designate unlabel anonymize deanonymize renameL renames detour	addV addH define		
SPPFs preserved or fail			removeV removeH undefine	
SPPFs refactored	unfold fold inline extract abridge unchain chain massage distribute factor deyaccify yaccify equate rassoc lassoc renameN clone concatT splitT	appear widen upgrade unite removeC	disappear	abstractize project concretize permute renameT splitN
SPPFs refactored or fail			addC narrow downgrade	redefine replace reroot
fail				inject

Table 1: The XBGF operator suite designed for convergence experiments [LZ09, LZ11] and updated here to the latest version of the GrammarLab. Columns of the table refer to the effects of the operators on the string language generated by a grammar; rows classify coupled effects on the SPPFs.

in all v_{ij} becoming equal to the corresponding v'_{ij}, such edges merge in the set. If such conjunctions represent ambiguities, this is disambiguation; if they represent parse views, it merges the views and makes them undistinguishable.

4 Grammar-based gardening

XBGF (standing for "transformations of BNF-like grammar formalism") was an operator suite for grammar programming originally developed for grammar recovery and convergence experiments [LZ09, LZ11] and used for various grammar maintenance tasks afterwards — e.g., for improving the quality and maturity of grammars in the Grammar Zoo [Zay15a, Zay15b]. It has operators like **eliminate**(n) that checks whether the given nonterminal n is referenced anywhere in the grammar, and if not, removes its definition harmlessly; or operators like **removeN**(x,y) that ensures that the nonterminal x is found in the grammar while y is not, and subsequently renames x to y; or even operators like **redefine**(p_k, p'_k) which removes all production rules p_k defining one nonterminal from the grammar and replaces them with rules p'_k defining the same nonterminal differently. These operators are relatively well-studied so that we can always make a claim about the effect that a transformation chain has on the language generated/accepted by the grammar. Originally [LZ11] XBGF operators were classified according to their preservation, increase, decrease or revision of the language within two semantics: the string semantics and the term semantics. The contribution of this section is their classification according to the coupled effect of the operators on the SPPFs — see Table 1 for the overview.

4.1 SPPFs preserved

The best kind of cotransformation is the trivial one where the initial transformation triggers no change in the linked artefacts.

4.1.1 Language-preserving operators

Many operators that preserve the (string) language associated with the grammar, also preserve the shared packed parse forests of the instances of this language. Consider, for instance, the **eliminate**(n) operator we have just introduced in the previous paragraph: essentially, it removes an unused construct. Since such a construct is unused in other production rules, it can never be reached from the root symbol, so it can also never occur in the graphs representing grammatically correct programs. Hence, any SPPF which was correct for grammar before the transformation, is still correct for the grammar with the unused part eliminated. Similarly, introducing a language construct that was not previously there and is not (yet) linked to the root, has no impact on the forests. The same argumentation holds for decorating operators that add/remove labels to/from rules of the grammar or their subexpressions, or rename them.

The last two operators seen in this cell on Table 1 are **vertical**(n) and **horizontal**(n) — they facilitate switching between a horizontal style of grammatical definitions (i.e., "`A ::= B | C;`") and a vertical one (i.e., "`A ::= B; A ::= C;`") — some grammatical frameworks distinguish between them, but never on an instance level, since a realisation of a disjunction commits to one particular branch. Hence, these operators also have no impact on SPPFs.

4.1.2 Language-extending operators

In the same way rearranging alternatives in production rules discussed in the previous section, has no impact on SPPFs, strict language extension operators like **addV**(p) and **addH**(p) have no impact on the forests. Since disjunctive clauses are not explicitly visible in SPPFs, any tree or forest derived with the original grammar, also conforms to the transformed one — the coupled instance transformation is trivial.

There is even one operator which is very invasive on a grammar level while being entirely harmless on the instance level — **define**(p) is a variant of **introduce**(p) that adds a definition of a nonterminal that *is used* in some parts in the grammar reachable from the top symbol. Having such nonterminals (called "bottom nonterminals") in a grammar is not a healthy practice and is in general considered a sign of bad quality since it signals incompleteness [LV01, SV00, Zay15a]. However, if we assume for the sake of simplicity that the default semantics for an undefined nonterminal is immediate failure (or parsing, generation, recognition or whatever the goal we need the grammar for), we may view **define**(p) as a language-extending (not a language-revising) operator. Thus, if we do somehow obtain a well-formed SPPF for such a grammar, it means it was constructed while *avoiding* the bottom nonterminal in question — hence, introducing it is no different than adding any other unreachable part we have seen so far and as such has no effect on the SPPFs.

4.2 SPPFs preserved, if possible

There are several cases when we do not know in advance whether the cotransformation of SPPFs will be possible: when it is, it is trivial.

4.2.1 Language-reducing operators

The operators **removeV**(p) and **removeH**(p) are the counterparts of **addV** and **addH** operators we have considered above, which remove alternatives instead of adding them. The effect of such a transformation on a given SPPF is easy to determine: if the alternative which is being removed, is exercised anywhere in the graph, the (co)transformation fails; if it is not, then no update of the forest is required.

Note that since all branches of the conjunctive clause are present in a given SPPF, their removal requires a (possibly failing) refactoring: hence, **removeC**(p) is considered later in Subsubsection 4.3.2.

The **undefine**(n) operator takes a valid nonterminal (defined and used within the grammar) and turns it into a bottom nonterminal (used yet not defined). It is a language reducing operator since its effect is a strict decrease in the number of possible correct programs: any parse graph containing a note related to the nonterminal n, becomes invalid. Hence, the cotransformation for it checks whether such a node is indeed found in the given SPPF: if yes, the transformation fails; if not, it immediately succeeds without updating the SPPF.

4.3 SPPFs refactored

In the next subsections we consider cases of less trivial cotransformations, when language instances have to change to preserve conformance.

4.3.1 Language-preserving operators

Many transformation operators that preserve the language associated with a grammar, still have some impact on the parse graphs. When the impact is easy to calculate in advance and thus encode the cotransformations as SPPF refactorings that are parametrised in the same way the grammar transformations are, we can run rewritings like **extract**(p) on both grammars and SPPFs.

Consider Figure 3(a). It shows a simple grammar of a language $\{a^n b^n c^n \mid n > 0\}$ with three conjunctive views: the first one (`a+ b+ c+`) being the most intuitive and hence the most suitable for expressing patterns to be matched on programs; the remaining two being used to parse the language (which is well-known to be context-sensitive, so we *need* the power of at least two conjuncts to recognise it precisely). In a sense, the last two conjuncts represent a recogniser and the first one specifies a parser [SJ10b, SC15]. When a transformation command **extract**(`AP::=a+;`) is executed, the effect on the grammar is apparent: a new nonterminal is introduced and two occurrences of its right hand side are replaced with it. The effect on an SPPF is also quite easy to calculate: the node with **a+** is replaced with a chain of two nodes (`AP` and `a+`); the incoming edges of the old node are connected as the incoming edges to the first one in the chain; the outgoing edges of the old node become the outgoing ones of the last in the chain (shown on Figure 3(b), changes in bold green). A slightly more complicated case

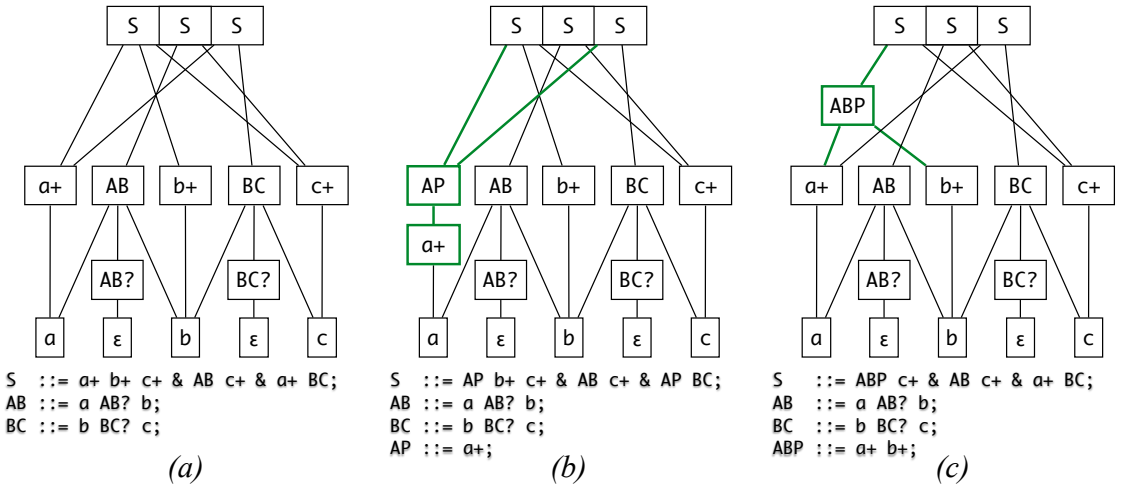


Figure 3: SPPF transformations coupled with extraction of a new nonterminal definition: **(a)** the original grammar and an SPPF of the term "`abc`"; **(b)** after applying **extract**(`AP::=a+;`); **(c)** after applying **extract**(`ABP::=a+ b+;`). The case of extracting one symbol is easier because an SPPF already has nodes for such derivations and they only need to be chained; when more than one symbol is present in the right hand side of a production rule being extracted, then a new node is introduced for all matched patterns of use.

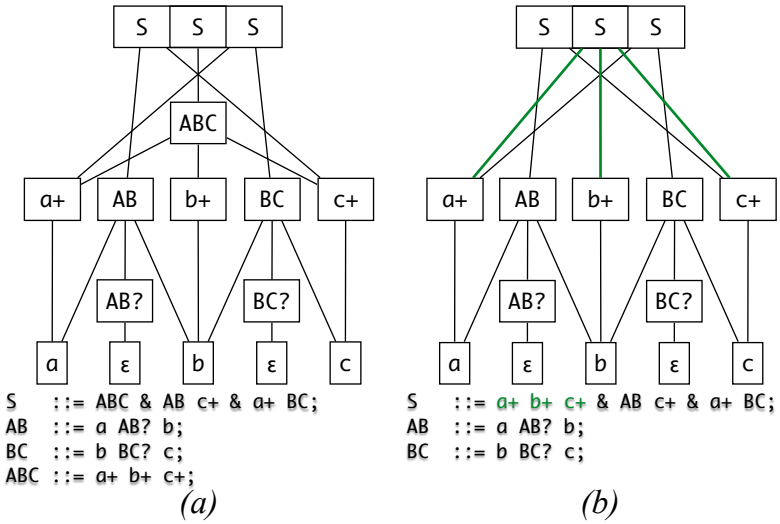


Figure 4: SPPF transformations coupled with inlining a nonterminal definition: **(a)** the original grammar and an SPPF of the term "`abc`"; **(b)** after applying **inline**(`ABC`). The inlining is fairly straightforward: the node in question is removed, and any previously incoming edge is replaced with the list of previously outgoing edges.

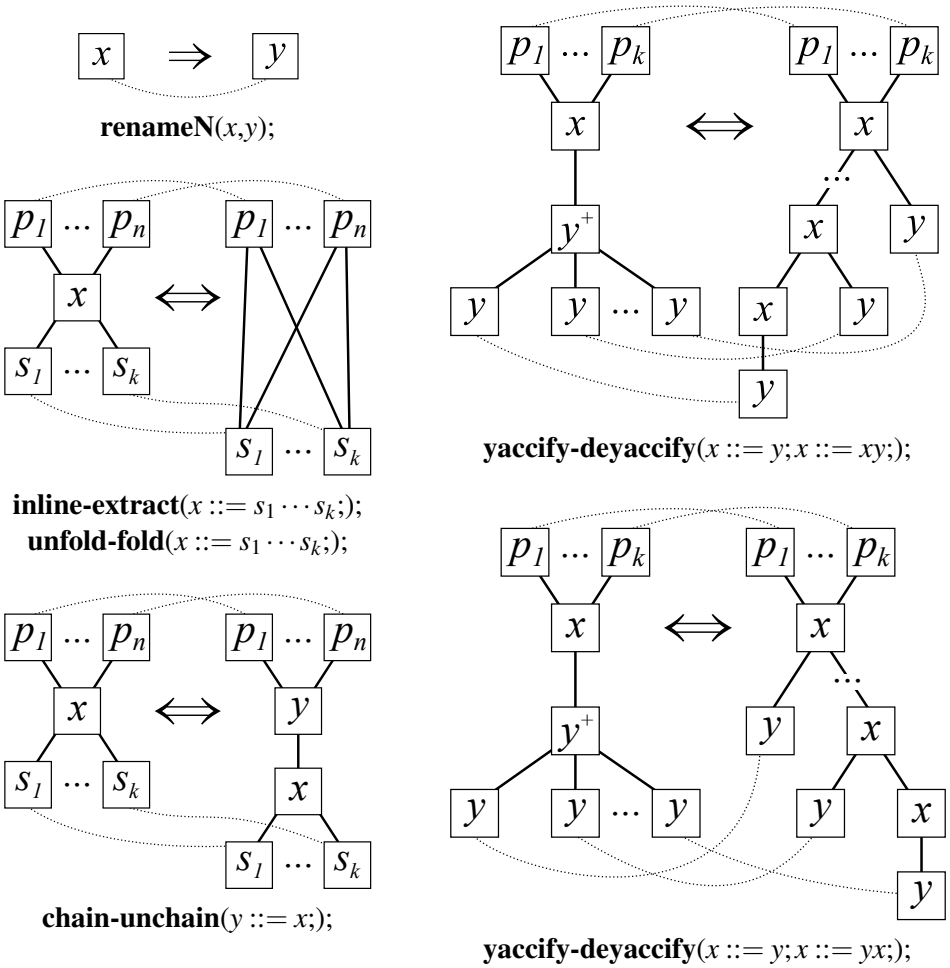


Figure 5: Graph transformations coupled to some of the language preserving grammar transformation operators: the graph on the left always represents a pattern being detected, the graph on the right defines the rewriting, dotted lines show correspondence between vertices that must not be broken by the transformation, so all incoming and outgoing edges around corresponding elements are to be preserved. **renameN** is very simple and intuitive: any node marked with x is replaced with a node marked with y while the node context is preserved. **inline-extract** shortcuts all incoming/outgoing edges to a node in question or introduces the node at their junction, depending on which direction we execute the cotransformation. **unfold-fold** works identically on the instance level, the differences are only observable on the grammar level. **chain-unchain** cuts a node in two, one of which inherits the incoming and the other one the outgoing edges. On the right, we show specifications for two ways of using **yaccify-deyaccify** to turn a flatter iteration-based SPPF into its deeper recursive counterpart, with left recursion variant on top and right recursion variant on the bottom. Other operators of this category from Table 1 are specified in a similar fashion.

is shown on Figure 3(c), where a new vertex needs to be created when we **extract**(`ABP::=a+ b+;`) because a symbol sequence `a+b+` did not correspond to any vertex in the old graph. For all vertices that had outgoing edges to both `a+` and `b+`, they got replaced by one edge to the new node. Figure 4 shows the opposite scenario of inlining a nonterminal in a grammar, coupled with "inlining" corresponding vertices in a graph by drawing edges through it.

By reusing our previous results on bidirectionalisation of grammar transformation operators [Zay12], we can significantly decrease the space needed for defining coupled instance transformations. In short, we know that if enough information is provided (e.g., the entire production rule being unfolded, not just a nonterminal name of its left hand side), then pairs of inverse operators can be treated as one bidirectional entity that allows forward and reverse applications. Indeed, as we have seen from Figure 3 and Figure 4, the output of one transformation has strong resemblance to the pattern responsible for applicability of the inverse transformation. An appropriately abstracted graph transformations of **inline-extract** and some other operators from Table 1 are shown on Figure 5.

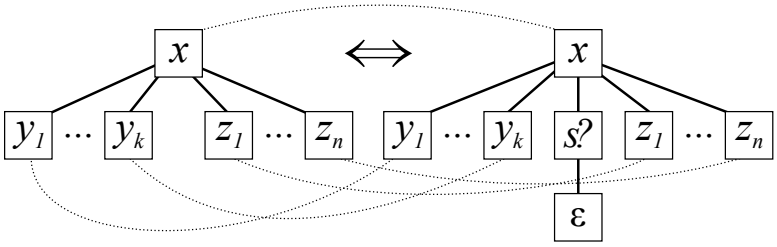
Conceptually the impact of applying these operators is hardly surprising: **chain** replaces a node with a chain of two nodes; **fold** does the same folding we have seen above with **extract**, but without introducing a new nonterminal and possibly in a limited scope; **rassoc** and **lassoc** replace an iterative production rule with a recursive right/left associative one and thus stretche a node with multiple children into an unbalanced binary subtree; **concatT** and **splitT** merge or unmerge leaves, etc.

4.3.2 Language-extending operators

Above we have considered grammar transformation operators that add disjunctive clauses to the grammar, obviously extending the associated language. In the case of extended context-free grammars (regular right hand side grammars) that allow metasyntactic sugar like optionals (x? effectively meaning $x|\varepsilon$) and regular closures (x^+ for transitive and x^* for reflexive transitive), the **widen**(e,e') operator is used to transform x? to x^* or x to x^+, together with the **appear**(p) operator that transforms ε to x? (effectively injecting an optional symbol). The coupled graph transformations for these cases usually boil down to inserting new vertices in the right places in order to keep the structural commitments up to date with the changed grammar.

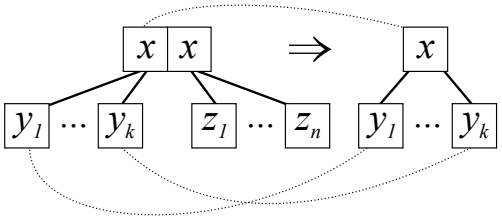
An even less trivial case of language extension is called "upgrading" and involves replacing a nonterminal by an expression that can be reduced to it. For instance, in `A ::= B C; D ::= B|E;` we can upgrade `B` in `A` (underlined) to `D`. Such a transformation increases the string language associated with a grammar, as well as rearranges the relations between nonterminals. The coupled transformation for SPPF is still simple and inserts an extra vertex for `D` between `A` and `B` (`E` is still not present in the SPPF).

The **removeC**(p) operator that eliminates a conjunct, formally also increases the underlying language since any extra conjunct is possibly an extra condition to be met, and dropping it makes the combination weaker. Technically the coupled SPPF transform that removes a conjunct is a disambiguation filter, but it is not useful to count it as such since the ambiguity being removed is explicit (recall Subsection 3.4). The graph transformations shown on Figure 6 define these coupled rewritings more formally. There are some interesting corner cases like partial coinciding of y_i and z_j which are outside the scope of this paper but should be researched in the future.



appear-disappear($x ::= y_1 \cdots y_k \ \underline{s?} \ z_1 \cdots z_n$);

unite(x, y); **widen**(x^+, x^*);



removeC($x ::= z_1 \cdots z_n$);

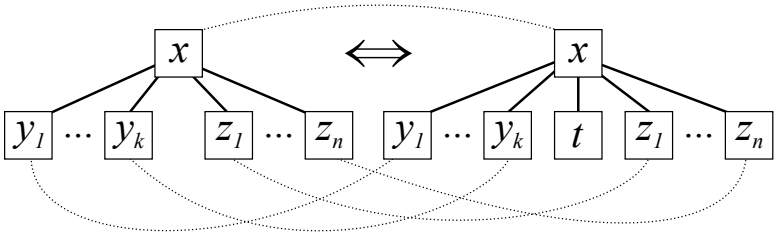
Figure 6: Graph transformations coupled to language-extending/-reducing grammar transformation operators, in the same format as used on the previous figure. **appear-disappear** matches the entire production rule in question but includes correspondences for each vertex so none of them can be damaged; the only modification is the addition of a new vertex with an empty value inside. **unite** adds all production rules of one nonterminal to another nonterminal, so all we need to do on the instance level is replace old nonterminal occurrences with the new one. **widen** is also unidirectional but changes only the type of one node, so the graph transformation specification looks very small. Finally, **removeC** removes a conjunctive clause from both the grammar and the graph defined by it. The transformation coupled to **upgrade** is not shown because it is identical to **extract** or **fold** which were already considered.

4.3.3 Language-reducing operators

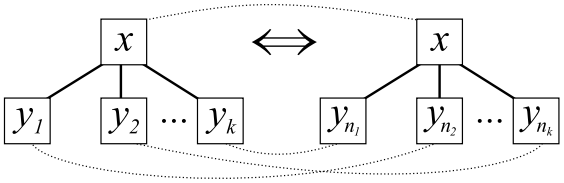
The **disappear**(p) operator is used to transform $x?$ or x^* to ε. The coupled transformation on SPPFs for it exists, but is subtly different from the ones being considered so far: it is inherently irreversible since if the SPPF in question actually contains the $x?$ with x as a child node, then that x is removed and lost. This is contrasting to folding/unfolding vertices and rearranging the edges around them.

4.3.4 Language-revising operators

The operators **abstractize**(p) and **concretize**(p) eliminate and introduce terminals from production rules (a common practice when mapping abstract syntax to concrete syntax, hence the



abstractize-concretize($x ::= y_1 \cdots y_k \; \underline{t} \; z_1 \cdots z_n$);
(only right to left:) **project**($x ::= y_1 \cdots y_k \; \underline{t} \; z_1 \cdots z_n$);



permute($x ::= y_1 \cdots y_k;, x ::= y_{n_1} \cdots y_{n_k};$);

Figure 7: Graph transformations coupled to language-revising grammar transformation operators, in the same format as used on the previous figures. The transformation on top uses the same pattern that we used before for **appear-disappear**, which was justified back then by the empty contents of the node and now by the node in question being terminal and hence having no children. The **permute** operator shuffles children of one node but preserves them, which is perfectly easy expressible by the correspondence relation of the graph transformation.

names). Since the terminals are present explicitly in the arguments, we can easily implement our coupled SPPF transformations by inserting leaves and connecting them to the appropriate places to the graph, or removing them. These transformations can have a big effect on the SPPF and are therefore more similar to the cotransformation from the previous paragraph. The **project** operator is a stronger version of **abstractize** or **disappear** that works on any symbol, but the transformation coupled with it, is the same: locate all the parts being removed from the grammar, remove them from the graph.

The rest of language revising operators are coupled with less invasive rearrangements of the parse graph: reordering edges (**permute**), updating the contents of the leaves (**renameT**) and splitting one nonterminal into several (**splitN**). We show two interesting graph transformations on Figure 7; the remaining ones are focused on the application scope instead of actual graph manipulation (**splitN**) or are identical to ones shown before (**renameT** is the same as **renameN** in all aspects).

4.4 SPPFs refactored, when possible

Cotransformations from the previous section were necessary but could never fail: they were applicable to all possible graphs. Let us now move on to cotransformations that could seem successful on the grammar level but fail on the instance level (causing the combination to fail).

4.4.1 Language-reducing operators

The **narrow** operator (the reverse of **widen** discussed above) and the **downgrade** operator (the reverse of **upgrade**) become simple parse graph rearrangements, if the constructs in the SPPF happen to correspond to the new grammar, and fail otherwise. For instance, if a "wider" option is found in the SPPF, we have no automated way to update it. Formally the graph transformation of **narrow** will look similar to the ones we have seen in **widen**, **renameN**, **unite**, etc, but to the best of our knowledge there is no graph rewriting framework that allows expressing the "if this pattern is matched, give up and report an error" which is the crucial other side of this cotransformation. The same holds for **upgrade** which is partially applicable folding.

The **addC** operator, on the other hand, shows us yet another class of cotransformations: namely, the one requiring reparsing. Indeed, if the first branch of the conjunctive clause of `S` from Figure 3 were to be introduced as a transformation step, we would need to reconnect the left subnode of `S` to the appropriate children, which formally corresponds to parsing. In the current prototype implementation we reuse the existing parser — to the best of our knowledge, other frameworks like TXL [DCMS02] do the same — instead of exploring possibly more efficient alternatives. Pure graph transformations do not seem to be the right solution here, because if we are confined by their use, parsers need to be propagated along some scenarios — which could certainly lead to curious exercises in combining continuation passing with pair grammars, but the idea looks over-engineered at first glance.

4.4.2 Language-revising operators

The most brutal among language revising operators: **redefine** that replaces an entire nonterminal definition with a different one; **replace** doing the same for arbitrary subexpressions; **reroot** that changes the starting symbol of the grammar, — all require reparsing as a part of their cotransformation steps.

4.5 Cotransformations destined to fail

Interestingly, there is one particular operator that is always doomed: **inject**(p) that works like **appear** but can insert any symbol anywhere in the grammar. In order to construct a coupled SPPF transformation for **inject**, we need to know how to connect the new node to its children, but this information is ultimately lacking from the operator parameters. The only cases where it could have worked, are already covered by other operators (e.g., injecting terminals is **concretize**, injecting possibly empty symbols is **appear**).

5 Related and future work

The disambiguation of context-free grammars is a problem that was claimed to be solved as early as 1995 [Che95] and is still actively researched in the 2010s [Sch10, BV11, VT13]. There is a lot of work on disambiguation, parse forest pruning and shaving, and it remains to be seen whether our approach can usefully complement similarly-minded techniques from that area such as van den Brand et al.'s implementation of disambiguation filters with term rewriting [BKMV03]. With

the technology from this paper we can perform "co-disambiguation" of grammars with forests. The obvious advantage of such a setup is that the presence of ambiguity on the instance level is trivially detectable while posing a long standing challenge on the grammar level.

Partial models [FSC12] allow for explicit modelling of uncertainty, of which ambiguity is an example of. There are many techniques accumulated in this area, allowing for decision making [AAFL96], clustering and classification [MPTV10], application of normal logic rules [LS02] — all in the face of uncertainty. A deeper investigation is needed to make strong claims about (in)compatibility of such approaches to transformation of ambiguous forests. Recent advancements in delta-lenses make them an attractive mechanism for this endeavour [DEPC16]. Walking this path will at least broaden the classic ambiguous grammar problem from purging uncertainty to tolerating it.

Stevenson [Ste15] and Cordy [BZ15] argue for 5 ways to approach the problem of transforming SPPFs: (1) abandon well-formedness, (2) use disjunction transformation semantics for conjuncts, (3) transform all views at once, (4) always trigger full reparse, (5) seamlessly work on canonic representation. Strictly speaking, there are also some methods to adjust the semantics of the grammar/metamodel itself — redefine what constitutes grammaticality; examples can be found in older literature, based on logical representation of ambiguous terms [Rae97], specificity rules for productions [Ken90], etc.

As said before, we are not the only ones trying to use computation models based on graphs instead of trees in software language engineering. It remains to be seen whether systematically using abstract syntax graphs [SL13] and general purpose graph transformation frameworks would be much different. As stated in the introduction, we deliberately focused our attention on methods that do *not* use higher order abstract syntax [PE88, DPS97, Chl08]. Our approach is closer to pair/triple grammars that are commonly used to define graph-to-graph rewritings [Pra71, Sch95]. We have noticed that in our case this way could have profited from an extension allowing to match separately on the incoming edges and the outgoing ones instead of matching the entire vertex, but we have not explored this option in full, since it seemed to fall in a somewhat unexplored space between established forms of node rewriting [ER97] and (hyper)edge rewriting [DKH97].

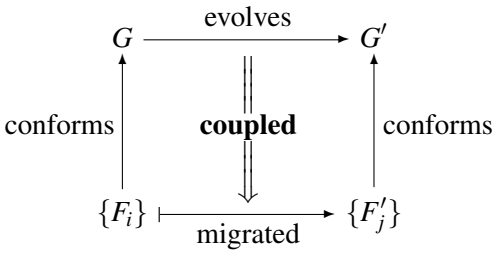
The approach to couple instance transformations to grammar transformations and not vice versa, that we decided to pursue, has its counterparts in other technological spaces such as modelware [GFD05] or XML [LL01] or databases [HHTH02], obviously with transformations of metamodels or schemata as the starting point. Cotransformations in general have been re-explained to some extent in this paper, but there exists a much more detailed introduction [Läm04].

Grammar mutations [Zay12, Zay14b] are systematic generalisations of grammar transformations used for this paper. There does not seem to be any fundamental problem in combining that generalisation with our couplings, but the implementation of coupled mutations remains future work.

The classification of coupled SPPF transformations from Table 1 corresponds to the two kinds of negotiated evolution: "adaptation through tolerance" when SPPFs are preserved and "through adjustment" when they are refactored [Zay14a].

6 Conclusion

In this paper, we have considered cotransformations of grammars together with shared packed parse forests defined by these grammars. That is, given a grammar G and a collection of graphs $\{F_i\}$ conforming to it, we can specify its evolution into an updated grammar G' such that the graphs are migrated to conform to G':



An implementation of a transformation operator suite was proposed. Each grammar change was coupled to one of the following: (1) no change in the parse graphs; (2) rearranging the graphs, implemented in triple graph-style rewritings; (3) introducing new elements to graphs based on operator arguments; (4) reparsing; (5) imminent failure. This classification is complementary to the previously existing ones based on preserving, increasing, reducing or revising the semantics chosen for the grammar.

The examples given in the paper mostly refer to concrete grammars in the context of parsing, but the research was done with software language engineering principles, which means that the contribution is applicable to co-evolution of *grammars in a broad sense* such as ontologies, API, DSLs, graph schemata, libraries, etc. We have used Boolean grammars as the underlying formalism due to their power to naturally represent non-context-free languages, ambiguous generalised parses and parse views in a uniform way. This is the first project involving coupled transformations of Boolean grammars.

The computation model proposed in this paper, can be used for formalisations and proofs of certain properties of transformation chains; for grammar-based convergence; for manipulating parse views and in general for tasks involving synchronous consistent changes to Boolean grammars and shared packed parse forests of grammatical instances. This is an area of rapidly growing interest in the software language engineering community, and its limits, as well as the extent of its usefulness, remain to be examined.

Bibliography

[AAFL96] B. Awerbuch, Y. Azar, A. Fiat, F. T. Leighton. Making Commitments in the Face of Uncertainty: How to Pick a Winner Almost Every Time. In *STOC*. Pp. 519–530. ACM, 1996.
DOI: 10.1145/237814.238000

[ASU85] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.

[BHKO02] M. G. J. van den Brand, J. Heering, P. Klint, P. A. Olivier. Compiling Language Definitions: The ASF+SDF Compiler. *ACM ToPLaS* 24(4):334–368, 2002.
DOI: 10.1145/567097.567099

[BJKO00] M. G. J. van den Brand, H. A. de Jong, P. Klint, P. A. Olivier. Efficient Annotated Terms. *Software: Practice & Experience* 30(3):259–291, 2000.

[BKMV03] M. van den Brand, S. Klusener, L. Moonen, J. J. Vinju. Generalized Parsing and Term Rewriting: Semantics Driven Disambiguation. *ENTCS* 82(3):575–591, 2003.
DOI: 10.1016/S1571-0661(05)82629-5

[BKV03] M. G. J. van den Brand, P. Klint, J. J. Vinju. Term Rewriting with Traversal Functions. *ACM ToSEM* 12(2):152–190, Apr. 2003.
DOI: 10.1145/941566.941568

[BL13] A. H. Bagge, R. Lämmel. Walk Your Tree Any Way You Want. In *ICMT*. LNCS 7909, pp. 33–49. Springer, 2013.
DOI: 10.1007/978-3-642-38883-5_3

[BMS05] G. M. Bierman, E. Meijer, W. Schulte. The Essence of Data Access in Cω. In *ECOOP*. LNCS 3586, pp. 287–311. Springer, 2005.
DOI: 10.1007/11531142_13

[BSVV02] M. van den Brand, J. Scheerder, J. J. Vinju, E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In *CC*. LNCS 2304, pp. 143–158. Springer, 2002.
DOI: 10.1007/3-540-45937-5_12

[BV11] B. Basten, J. J. Vinju. Parse Forest Diagnostics with Dr. Ambiguity. In *SLE*. LNCS 6940, pp. 283–302. Springer, 2011.
DOI: 10.1007/978-3-642-28830-2_16

[BZ15] A. H. Bagge, V. Zaytsev. Open and Original Problems in Software Language Engineering 2015 Workshop Report. *SIGSOFT SE Notes* 40(3):32–37, 2015.
DOI: 10.1145/2757308.2757313

[Can62] D. G. Cantor. On the Ambiguity Problem of Backus Systems. *Journal of the ACM* 9(4):477–479, 1962.

[Che95] B. S. N. Cheung. Ambiguity in Context-Free Grammars. In *SAC*. Pp. 272–276. ACM, 1995.
DOI: 10.1145/315891.315991

[Chl08] A. Chlipala. Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In *ICFP*. Pp. 143–156. ACM, 2008.
DOI: 10.1145/1411204.1411226

[Cho59] N. Chomsky. On Certain Formal Properties of Grammars. *Information and Control* 2(2):137–167, 1959.
DOI: 10.1016/S0019-9958(59)90362-6

[CV11] A. Cunha, J. Visser. Transformation of Structure-shy Programs with Application to XPath Queries and Strategic Functions. *SCP* 76(6):516–539, 2011.
DOI: 10.1016/j.scico.2010.01.003

[DCMS02] T. R. Dean, J. R. Cordy, A. J. Malton, K. A. Schneider. Grammar Programming in TXL. In *SCAM*. IEEE Computer Society, 2002.
DOI: 10.1109/SCAM.2002.1134109

[DEPC16] Z. Diskin, R. Eramo, A. Pierantonio, K. Czarnecki. Introducing Delta-Lenses with Uncertainty. In *BX*. 2016. In print.

[DFH95] J. Despeyroux, A. P. Felty, A. Hirschowitz. Higher-Order Abstract Syntax in Coq. In *TLCA*. LNCS 902, pp. 124–138. Springer, 1995.
DOI: 10.1007/BFb0014049

[DKH97] F. Drewes, H.-J. Kreowski, A. Habel. Hyperedge Replacement Graph Grammars. In *Handbook of Graph Grammars and Computing by Graph Transformation*. Pp. 95–162. World Scientific Publishing Co., Inc., 1997.

[DPS97] J. Despeyroux, F. Pfenning, C. Schürmann. Primitive Recursion for Higher-Order Abstract Syntax. In *TLCA*. LNCS 1210, pp. 147–163. Springer, 1997.
DOI: 10.1007/3-540-62688-3_34

[Ear68] J. Earley. *An Efficient Context-Free Parsing Algorithm*. PhD thesis, Carnegie-Mellon, Aug. 1968.

[ER97] J. Engelfriet, G. Rozenberg. Node Replacement Graph Grammars. In *Handbook of Graph Grammars and Computing by Graph Transformation*. Pp. 1–94. World Scientific Publishing Co., Inc., 1997.

[For04] B. Ford. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *POPL*. Pp. 111–122. ACM, 2004.
DOI: 10.1145/964001.964011

[FSC12] M. Famelis, R. Salay, M. Chechik. Partial Models: Towards Modeling and Reasoning with Uncertainty. In *ICSE*. Pp. 573–583. IEEE, 2012.
DOI: 10.1109/ICSE.2012.6227159

[GFD05] T. Gîrba, J.-M. Favre, S. Ducasse. Using Meta-Model Transformation to Model Software Evolution. *ENTCS* 137(3):57–64, 2005.
DOI: 10.1016/j.entcs.2005.07.005

[HHTH02] J. Henrard, J.-M. Hick, P. Thiran, J.-L. Hainaut. Strategies for Data Reengineering. In *WCRE*. Pp. 211–220. IEEE Computer Society, 2002.

[Hin00] R. Hinze. A New Approach to Generic Functional Programming. In *POPL*. Pp. 119–132. ACM, 2000.
DOI: 10.1145/325694.325709

[JAB+06] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, P. Valduriez. ATL: a QVT-like Transformation Language. In *OOPSLA*. Pp. 719–720. ACM, 2006.
DOI: 10.1145/1176617.1176691

[JJ97] P. Jansson, J. Jeuring. PolyP — A Polytypic Programming Language Extension. In *POPL*. Pp. 470–482. ACM Press, 1997.
DOI: 10.1145/263699.263763

[Kay07] M. Kay. XSL Transformations (XSLT) Version 2.0. *W3C Recommendation*, 23 January 2007. http://www.w3.org/TR/2007/REC-xslt20-20070123.

[Ken90] R. Kennaway. The Specificity Rule for Lazy Pattern-Matching in Ambiguous Term Rewrite Systems. In *ESOP*. LNCS 432, pp. 256–270. Springer, 1990.
DOI: 10.1007/3-540-52592-0_68

[KLV02] J. Kort, R. Lämmel, C. Verhoef. The Grammar Deployment Kit — System Demonstration. *ENTCS* 65(3):117–123, 2002.
DOI: 10.1016/S1571-0661(04)80430-4

[KSV09] P. Klint, T. van der Storm, J. J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *SCAM*. Pp. 168–177. IEEE Computer Society, 2009.
DOI: 10.1109/SCAM.2009.28

[KV94] P. Klint, E. Visser. Using Filters for the Disambiguation of Context-Free Grammars. In *Proceedings of the ASMICS Workshop on Parsing Theory*. Pp. 1–20. 1994.

[KV10] L. C. L. Kats, E. Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *OOPSLA*. Pp. 444–463. ACM, 2010.
DOI: 10.1145/1869459.1869497

[Lä05] R. Lämmel. The Amsterdam Toolkit for Language Archaeology. *ENTCS* 137(3):43–55, 2005.
DOI: 10.1016/j.entcs.2005.07.004

[Läm04] R. Lämmel. Transformations Everywhere. *SCP* 52:1–8, 2004.
DOI: 10.1016/j.scico.2004.03.001

[LJ04] R. Lämmel, S. L. P. Jones. Scrap More Boilerplate: Reflection, Zips, and Generalised Casts. In *ICFP*. Pp. 244–255. ACM, 2004.
DOI: 10.1145/1016850.1016883

[LL01] R. Lämmel, W. Lohmann. Format Evolution. In *RETIS*. Volume 155, pp. 113–134. OCG, 2001.

[LS02] Y. Loyer, U. Straccia. The Well-Founded Semantics in Normal Logic Programs with Uncertainty. In *FLOPS*. LNCS 2441, pp. 152–166. Springer, 2002.
DOI: 10.1007/3-540-45788-7_9

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

[LV03] R. Lämmel, J. Visser. A Strafunski Application Letter. In *PADL*. LNCS 2562, pp. 357–375. Springer, 2003.
DOI: 10.1007/3-540-36388-2_24

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

[LZ09] R. Lämmel, V. Zaytsev. An Introduction to Grammar Convergence. In *iFM*. LNCS 5423, pp. 246–260. Springer, Feb. 2009.
DOI: 10.1007/978-3-642-00255-7_17

[LZ11] R. Lämmel, V. Zaytsev. Recovering Grammar Relationships for the Java Language Specification. *SQJ* 19(2):333–378, Mar. 2011.
DOI: 10.1007/s11219-010-9116-5

[MA03] A. Momigliano, S. Ambler. Multi-level Meta-reasoning with Higher-Order Abstract Syntax. In *FoSSaCS*. LNCS 2620, pp. 375–391. Springer, 2003.
DOI: 10.1007/3-540-36576-1_24

[MPTV10] N. R. D. Matteo, S. Peroni, F. Tamburini, F. Vitali. Of Mice and Terms: Clustering Algorithms on Ambiguous Terms in Folksonomies. In *SAC*. Pp. 844–848. ACM, 2010.
DOI: 10.1145/1774088.1774262

[Okh01] A. Okhotin. Conjunctive Grammars. *Journal of Automata, Languages and Combinatorics* 6(4):519–535, 2001.

[Okh04] A. Okhotin. Boolean Grammars. *Information and Computation* 194(1):19–48, 2004.
DOI: 10.1016/j.ic.2004.03.006

[Okh13] A. Okhotin. Conjunctive and Boolean Grammars: The True General Case of the Context-Free Grammars. *Computer Science Review* 9:27–59, 2013.
DOI: 10.1016/j.cosrev.2013.06.001

[PE88] F. Pfenning, C. Elliott. Higher-Order Abstract Syntax. In *PLDI*. Pp. 199–208. ACM, 1988.
DOI: 10.1145/53990.54010

[PF11] T. Parr, K. Fisher. LL(*): the foundation of the ANTLR parser generator. In *PLDI*. Pp. 425–436. ACM, 2011.
DOI: 10.1145/1993498.1993548

[Pra71] T. W. Pratt. Pair Grammars, Graph Languages and String-to-graph Translations. *Journal of Computer and System Sciences* 5(6):560–595, 1971.
DOI: http://dx.doi.org/10.1016/S0022-0000(71)80016-8

[Rae97] I. Raeva. Semantic Interpretation of Ambiguous Statements, Represented in a Logical Form. In *Developments in Language Theory*. Pp. 529–537. Aristotle University of Thessaloniki, 1997.

[RHB01] C. Röckl, D. Hirschkoff, S. Berghofer. Higher-Order Abstract Syntax with Induction in Isabelle/HOL: Formalizing the π-Calculus and Mechanizing the Theory of Contexts. In *FoSSaCS*. LNCS 2030, pp. 364–378. Springer, 2001.
DOI: 10.1007/3-540-45315-6_24

[SC15] A. Stevenson, J. R. Cordy. Parse Views with Boolean Grammars. *SCP* 97(1):59–63, 2015.
DOI: 10.1016/j.scico.2013.11.007

[Sch95] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *20th International Workshop on Graph-Theoretic Concepts in Computer Science*. Pp. 151–163. Springer, 1995.
DOI: 10.1007/3-540-59071-4_45

[Sch10] S. Schmitz. An Experimental Ambiguity Detection Tool. *SCP* 75(1-2):71–84, 2010.
DOI: 10.1016/j.scico.2009.07.002

[SJ05] E. Scott, A. Johnstone. Generalized Bottom Up Parsers with Reduced Stack Activity. *Computer Journal* 48(5):565–587, 2005.
DOI: 10.1093/comjnl/bxh102

[SJ10a] E. Scott, A. Johnstone. GLL Parsing. *ENTCS* 253(7):177–189, 2010.
DOI: 10.1016/j.entcs.2010.08.041

[SJ10b] E. Scott, A. Johnstone. Recognition is not Parsing — SPPF-style Parsing from Cubic Recognisers. *SCP* 75(1-2):55–70, 2010.
DOI: 10.1016/j.scico.2009.07.001

[SL13] B. C. d. S. Oliveira, A. Löh. Abstract Syntax Graphs for Domain Specific Languages. In *PEPM*. Pp. 87–96. ACM, 2013.
DOI: 10.1145/2426890.2426909

[Ste15] A. Stevenson. *Source Transformations with Boolean Grammars*. PhD thesis, Queen's University, 2015.

[SV00] M. P. A. Sellink, C. Verhoef. Development, Assessment, and Reengineering of Language Descriptions. In *CSMR*. Pp. 151–160. IEEE Computer Society, 2000.

[Tom85] M. Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.

[VBT98] E. Visser, Z.-E.-A. Benaissa, A. P. Tolmach. Building Program Optimizers with Rewriting Strategies. In *ICFP*. Pp. 13–26. ACM, 1998.
DOI: 10.1145/289423.289425

[Vis97] E. Visser. *Scannerless Generalized-LR Parsing*. PhD thesis, UvA, 1997.

[Vis01] E. Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *RTA*. LNCS 2051, pp. 357–362. Springer, 2001.
DOI: 10.1007/3-540-45127-7_27

[VT13] N. Vasudevan, L. Tratt. Detecting Ambiguity in Programming Language Grammars. In *SLE*. LNCS 8225, pp. 157–176. Springer, 2013.
DOI: 10.1007/978-3-319-02654-1_9

[Zay12] V. Zaytsev. Language Evolution, Metasyntactically. *EC-EASST; BX* 49, 2012.

[Zay13] V. Zaytsev. Modelling Robustness with Conjunctive Grammars. In *SATToSE*. July 2013.

[Zay14a] V. Zaytsev. Negotiated Grammar Evolution. *JOT; XM* 13(3):1:1–22, July 2014.
DOI: 10.5381/jot.2014.13.3.a1

[Zay14b] V. Zaytsev. Software Language Engineering by Intentional Rewriting. *EC-EASST; SQM* 65, Mar. 2014.

[Zay15a] V. Zaytsev. Grammar Maturity Model. In *ME*. CEUR 1331, pp. 42–51. CEUR-WS.org, Feb. 2015.

[Zay15b] V. Zaytsev. Grammar Zoo: A Corpus of Experimental Grammarware. *SCP* 98:28–51, Feb. 2015.
DOI: 10.1016/j.scico.2014.07.010

[ZB14] V. Zaytsev, A. H. Bagge. Parsing in a Broad Sense. In *MoDELS*. LNCS 8767, pp. 50–67. Springer, 2014.
DOI: 10.1007/978-3-319-11653-2_4