

Grammar Adaptation

Ralf Lämmel

CWI, Kruislaan 413, NL-1098 SJ Amsterdam
Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam
Email: ralf@cwi.nl
WWW: <http://www.cwi.nl/~ralf/>

Abstract. We employ transformations for the adaptation of grammars. Grammars need to be adapted in grammar development, grammar maintenance, grammar reengineering, and grammar recovery. Starting from a few fundamental transformation primitives and combinators, we derive an operator suite for grammar adaptation. Three groups of operators are identified, namely operators for refactoring, construction and destruction. While refactoring is semantics-preserving in the narrow sense, transformations for construction and destruction require the consideration of relaxed notions of semantics preservation based on other grammar relations than equality of generated languages. The consideration of semantics and accompanying preservation properties is slightly complicated by the fact that we cannot insist on reduced grammars.

1 Introduction

Grammar adaptation We consider formal transformations facilitating the stepwise adaptation of grammars. These transformations model common schemes of restructuring (e.g., based on fold and unfold), and local changes (e.g., restriction, generalisation or removal of phrases). We focus on grammar transformations mimicking simple adaptation steps which are performed by “grammar programmers” in grammar development manually otherwise. The amount of grammar programming should not be underestimated. Our approach is relevant for grammar development, maintenance, reengineering, and recovery.

Grammar recovery The transformations formalised in the present paper proved to be valuable in actual grammar recovery projects. Grammar recovery is concerned with the derivation of a language’s grammar from some available resource such as a semi-formal language reference. Grammar transformations can be used to correct, to complete, and to restructure the raw grammar extracted from the resource in a stepwise manner. Grammar recovery is of prime importance for software reengineering—in particular for automated software renovation [5, 3]. The Y2K-problem or the Euro-conversion problem are some well-recognized reengineering problems. Tool support in software renovation, e.g., for grammar-based software modification tools, relies on grammars for the languages at hand. Suitable grammars are often not available, since one might be faced with ancient languages including all kinds of COBOL dialects, or with in-house languages

and language extensions. Thus, there is a need for grammar recovery. In one particular recovery project, we obtained the first publicly available, high-quality COBOL grammar [8]. The effort for this undertaking was relatively small compared to other known figures (refer to [9] for details). The use of grammar transformations was crucial to make this process accessible for tool support, traceable regarding the many steps and decisions involved, and measurable to allow for formal reasoning and claims like correctness and completeness. A global account on grammar recovery and grammar (re-) engineering from a software engineering perspective is given in [14, 9].

2.7.8 REDEFINES-clause

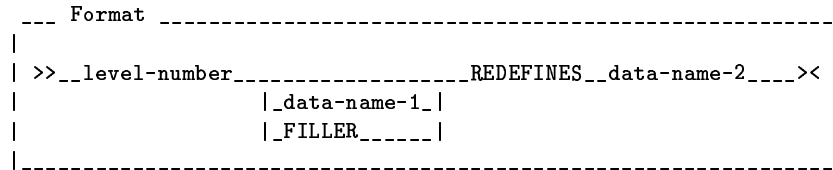


Fig. 1. An incorrect syntax diagram from [7]

Sample adaptation In recovering the VS COBOL II grammar [8] from the IBM reference [7], we were faced with a surprising number of problems regarding the syntax definition contained in the reference.¹ There were errors and omissions in the narrow sense. The syntax definition also suffered from the use of informal comments to regulate syntax. It is actually the lack of (use of) formal methods that causes such important documents to be incorrect and incomplete. One incorrectness is shown in the syntax diagram in Fig. 1. The diagram is supposed to define the syntax of a REDEFINES-clause which is an optional clause of a COBOL data item. For convenience, we also define the piece of syntax from the diagram in extended BNF notation:²

```

REDEFINES-clause ::=
    level-number (data-name | "FILLER")? "REDEFINES" data-name

```

The diagram or its integration in the overall grammar is incorrect because the diagram does not just define the structure of a REDEFINES-clause, but rather the structure of a data item *with* a REDEFINES-clause. A proper REDEFINES-clause is simply of the form of "REDEFINES" data-name. The problem can easily be recognized by parsing approved code if the (prototype) parser is derived directly from the diagrams as described in [9]. To correct the diagram or the grammar resp., we need to delete the phrase which does not belong to a proper REDEFINES-clause. Using a corresponding transformation operator for grammar adaptation, this removal can be expressed as follows:

¹ About 300 transformation steps were needed to derive [8] from [7].

² We enclose terminals in double quotes ("..."). "|" separates alternatives. "?" is a postfix operator for optionals. Grouping is done with "(...)".

```
delete level-number (data-name | "FILLER")? in {REDEFINES-clause}
```

Benefit of the transformational approach Transformational grammar adaptation improves on ad-hoc manual adaptation in several respects. Transformations add traceability since changes can be recorded. In actual grammar development, maintenance, recovery etc., it is important to state clearly what changes were needed. The resulting adaptation scripts are also reusable in part for grammars of dialects, or if some adaptation decisions are altered. The properties of transformation operators immediately qualify a certain adaptation accordingly. For those operators, which are not semantics-preserving in the narrow sense, we provide corresponding relaxed notions characterising the impact of the operators. In the above sample, we non-ambiguously identify the phrase to be removed. Also, since the transformation is performed in a focus, we document the intended area for the local change. In grammar programming, it is indeed important to understand the impact that a change might have, since grammars serve as contracts for language tools [6]. In general, we envision that our approach to grammar adaptation contributes to completeness and correctness claims for grammars.

Structure of the paper In Section 2, we propose a variant of context-free grammars particularly useful to cope with evolving grammars. In Section 3, binary grammar relations and induced transformation properties are studied to enable formal reasoning for transformational grammar adaptation. The contribution of this section is that we go beyond semantics-preservation in the narrow sense. In Section 4, a transformation framework is developed. It offers primitives and combinators for grammar transformation. In Section 5, operators for grammar adaptation, e.g., the operator `delete` used in the example above, are derived in the transformation framework. The paper is concluded in Section 6.

Acknowledgement This work was supported, in part, by *NWO*, in the project “*Generation of Program Transformation Systems*”. I am grateful for advice by the FME'01 anonymous referees. I am also grateful to Jeremy Gibbons, Jan Heering, Merijn de Jonge, Paul Klint, Chris Verhoef, Joost Visser, Guido Wachsmuth, and Dave Wile for encouraging discussions, and for helpful suggestions. Some results from the present paper have been presented at the 54th IFIP WG2.1 (“Algorithmic Languages and Calculi”) meeting in Blackheath, London, April, 3th–7th, 2000, and at the 2nd Workshop Software-Reengineering, Bad Honnef, May, 11th–12th, 2000.

2 Grammar fragments

We derive a simple variant of context-free grammars as the object language for grammar transformations. We call this variant *grammar fragments*. The idea of grammar adaptation implies that we cannot assume reduced context-free grammars. We rather have to cope with evolving grammars.

2.1 Standard context-free grammars

A context-free grammar G is a quadruple $\langle N, T, s, P \rangle$, where N and T are the disjoint finite sets of nonterminals resp. terminals, $s \in N$ is called start symbol, and P is a finite set of productions or (context-free) rules with $P \subset N \times (N \cup T)^*$. A production $\langle n, u \rangle \in P$ with $n \in N$ and $u \in (N \cup T)^*$ is also written as $n \rightarrow u$. It is common to assume reduced grammars, that is, each production can be used in some derivation of a terminal string from the start symbol.

2.2 Deviation

We slightly deviate from the standard. We do not insist on the identification of a start symbol s because this is definitely not sensible for incomplete grammars. Furthermore, we do not explicitly declare nonterminals and terminals. A grammar is represented just by its productions. Nonterminals and terminals originate from different universes of grammar symbols, that is, \mathcal{U}_N for nonterminals, and \mathcal{U}_T for terminals. We call the resulting deviation *grammar fragments*. We do not require grammar fragments to be terminated, that is, there can be nonterminals which are used, but not defined. We call these nonterminals *bottom nonterminals*. The domains of all context-free productions \mathcal{R} (or rules for short) and grammar fragments \mathcal{G} are defined as follows:³

$$\begin{aligned}\mathcal{R} &= \mathcal{U}_N \times (\mathcal{U}_N \cup \mathcal{U}_T)^* \\ \mathcal{G} &= \mathcal{P}_{fin}(\mathcal{R})\end{aligned}$$

We use γ possibly indexed or primed to range over \mathcal{G} . The terms $\mathcal{T}(\gamma) \subset \mathcal{U}_T$ and $\mathcal{N}(\gamma) \subset \mathcal{U}_N$ are used to denote terminals and nonterminals of γ . These and other relevant sets of grammar symbols are defined in Fig. 2.

$$\begin{aligned}\mathcal{D}(\gamma) &= \{n \in \mathcal{U}_N \mid n \rightarrow u \in \gamma, u \in (\mathcal{U}_N \cup \mathcal{U}_T)^*\} \\ \mathcal{U}(\gamma) &= \{n' \in \mathcal{U}_N \mid n \rightarrow u n' v \in \gamma, n \in \mathcal{U}_N, u, v \in (\mathcal{U}_N \cup \mathcal{U}_T)^*\} \\ \mathcal{N}(\gamma) &= \mathcal{D}(\gamma) \cup \mathcal{U}(\gamma) \\ \mathcal{T}(\gamma) &= \{t \in \mathcal{U}_T \mid n \rightarrow u t v \in \gamma, n \in \mathcal{U}_N, u, v \in (\mathcal{U}_N \cup \mathcal{U}_T)^*\} \\ \perp(\gamma) &= \mathcal{U}(\gamma) \setminus \mathcal{D}(\gamma) \\ \top(\gamma) &= \mathcal{D}(\gamma) \setminus \mathcal{U}(\gamma) \\ \emptyset(\gamma) &= \{n \in \mathcal{N}(\gamma) \mid \exists w \in (\perp(\gamma) \cup \mathcal{T}(\gamma))^*. n \Rightarrow_{\gamma}^* w\}\end{aligned}$$

Fig. 2. Defined nonterminals $\mathcal{D}(\gamma)$, used nonterminals $\mathcal{U}(\gamma)$, nonterminals $\mathcal{N}(\gamma)$, terminals $\mathcal{T}(\gamma)$, bottom nonterminals $\perp(\gamma)$, top nonterminals $\top(\gamma)$, looping nonterminals $\emptyset(\gamma)$

Bottom nonterminals capture some important properties of a grammar fragment γ . In grammar development, a bottom nonterminal might correspond to

³ $\mathcal{P}_{fin}(x)$ denotes the finite subsets of x .

a nonterminal lacking a definition. Alternatively, a bottom nonterminal might indicate a connectivity problem, that is, the intended definition is given with a different nonterminal symbol on the left-hand side. There is also the notion of *top nonterminals* which is somewhat dual to the notion of bottom nonterminals. As Fig. 2 details, the set $\top(\gamma)$ of top nonterminals consists of all nonterminals defined but not used in γ . A start symbol usually meets this condition.⁴ An incomplete and/or incorrect grammar usually exhibits several such top nonterminals. Thus, in a sense, top nonterminals provide an indication to what extent a grammar fragment is connected. The set $\emptyset(\gamma)$ of looping nonterminals which is also introduced in Fig. 2 will be explained later.

2.3 Semantics

Terminal strings The common semantics for context-free grammars is based on derivation (\Rightarrow etc.). The language $\mathcal{L}(G)$ generated by a common context-free grammar $G = \langle N, T, s, P \rangle$ is usually defined as follows:

$$\mathcal{L}(G) = \{w \in T^* \mid s \Rightarrow_G^+ w\},$$

i.e., as the set of terminal strings derivable from the dedicated start symbol. Of course, one can also consider the terminal strings derivable from an arbitrary nonterminal n . We are going to denote the semantics of n w.r.t. a grammar fragment γ as $\llbracket n \rrbracket_\gamma$. Adopting the idea of generated terminal strings, we get a first approximation of the ultimate denotation of n , namely:

$$\llbracket n \rrbracket_\gamma \supseteq \{w \in \mathcal{T}(\gamma)^* \mid n \Rightarrow_\gamma^+ w\}$$

It is easy to acknowledge that terminal strings are not sufficient since grammar fragments are not necessarily terminated. Productions which contain bottom nonterminals can never contribute to the set of derivable terminal strings. The sets of terminal strings generated by some defined nonterminals will even be empty. Consider, for example, the incomplete grammar γ_1 in Fig. 3 (the other content of the figure will be explained later). Indeed, the defined nonterminal s (the definition of which refers to the bottom nonterminal b) generates the empty set of terminal strings. This is not convenient for a semantics because a grammar fragment is not fully reflected by derivable terminal strings. Thus, terminal strings can only provide a lower bound for the ultimate denotation $\llbracket n \rrbracket$ of a nonterminal n .

Sentential forms Instead of pure terminal strings, sentential forms can be taken into account. As we will see, sentential forms correspond to an upper bound for the ultimate denotation. Thus, we have the following:

$$\llbracket n \rrbracket_\gamma \subseteq \{w \in (\mathcal{N}(\gamma) \cup \mathcal{T}(\gamma))^* \mid n \Rightarrow_\gamma^* w\}$$

⁴ A relaxed definition of top nonterminals is favourable: A top nonterminal might be used in the rules for the nonterminal itself. This definition is useful to cope with recursive start symbols and to provide a strong criterion for safe elimination of nonterminals.

Sentential forms are too sensitive. They do not even provide a basis to state the semantics preservation of fold/unfold modulations. Such modulations are obviously very useful in refactoring a grammar.

$$\begin{array}{ccc}
\gamma_1 & & \gamma_2 \\
s \rightarrow ab & \llbracket b \rrbracket_{\gamma_1}^\emptyset = \emptyset = \llbracket b \rrbracket_{\gamma_2}^\emptyset = \emptyset & s \rightarrow ab \\
a \rightarrow \epsilon & \llbracket b \rrbracket_{\gamma_1}^{\perp(\gamma_1)} = \{b\} \supset \llbracket b \rrbracket_{\gamma_2}^{\perp(\gamma_2)} = \emptyset & a \rightarrow \epsilon \\
a \rightarrow \text{"A"} a & \llbracket b \rrbracket_{\gamma_1}^{\perp(\gamma_1) \cup \emptyset(\gamma_1)} = \{b\} \subset \llbracket b \rrbracket_{\gamma_2}^{\perp(\gamma_2) \cup \emptyset(\gamma_2)} = \{b, \text{"B"} b, \dots\} & a \rightarrow \text{"A"} a \\
& & b \rightarrow \text{"B"} b
\end{array}$$

Fig. 3. Illustration of incomplete grammars

Observable nonterminals and sentential forms We need to restrict the sentential forms so that only particular nonterminals are *observable*. It is sensible to require at least the bottom nonterminals to be observable. Then we can still derive all “interesting” sentential forms for γ_1 in Fig. 3. To be slightly more flexible regarding observable nonterminals, we consider an augmented variant of the semantic function with a superscript ψ for the observable nonterminals:

$$\llbracket n \rrbracket_{\gamma}^{\psi} = \text{def } \{w \in (\psi \cup \mathcal{T}(\gamma))^* \mid n \Rightarrow_{\gamma}^* w\} \text{ where } \psi \subseteq \mathcal{N}(\gamma)$$

We can read $\llbracket n \rrbracket_{\gamma}^{\psi}$ as the semantics of n according to γ assuming the nonterminals in ψ are observable. We call the corresponding sentential forms observable, too. The semantics restricted to sentential forms consisting solely of bottom nonterminals and terminals is then denoted by $\llbracket n \rrbracket_{\gamma}^{\perp(\gamma)}$. These sentential forms provide a better lower bound for the ultimate semantics than the one we had before (i.e., terminal strings). Thus, we have the following:

$$\llbracket n \rrbracket_{\gamma} \supseteq \llbracket n \rrbracket_{\gamma}^{\perp(\gamma)}$$

We have to explain why observing bottom nonterminals only leads to a lower bound. Consider grammar γ_2 in Fig. 3 for that purpose. Although, b is defined, b does not generate strings over $\perp(\gamma_2) \cup \mathcal{T}(\gamma_2)$. This is caused by the particular definition of b in γ_2 , that is, the definition is *looping* because there is no base case for b .

Looping nonterminals We resolve the above problem by observing also the set $\emptyset(\gamma)$ of all looping nonterminals as defined in Fig. 2. Finally, we are in the position to define the denotation of n w.r.t. γ in the ultimate way as follows:

$$\llbracket n \rrbracket_{\gamma} = \text{def } \llbracket n \rrbracket_{\gamma}^{\perp(\gamma) \cup \emptyset(\gamma)}$$

This semantics assumes the minimum of observable nonterminals without causing the denotation of some nonterminal to be empty.

3 Formal reasoning

We are interested in comparing grammars, and in characterising the properties of grammar transformations. For this purpose, we define certain relations on \mathcal{G} starting from the equivalence of grammar fragments. Finally, the grammar relations are employed for defining several forms of semantics-preservation.

3.1 Equivalent grammars

Definition 1 (Equivalence (\equiv)). γ and γ' are equivalent ($\gamma \equiv \gamma'$) if:

1. $\mathcal{N}(\gamma) = \mathcal{N}(\gamma')$,
2. $\forall n \in \mathcal{N}(\gamma). \llbracket n \rrbracket_{\gamma} = \llbracket n \rrbracket_{\gamma'}$,
3. $\perp(\gamma) = \perp(\gamma')$, $\mathcal{D}(\gamma) = \mathcal{D}(\gamma')$.⁵

This relation is useful to characterize fold/unfold modulations. All grammar relations will be exemplified in Fig. 4. As for equivalence, it holds $\gamma_1 \equiv \gamma_2$. Note that γ_1 and γ_2 are indeed related via unfold: The definition of y is unfolded in the first rule of γ_2 . The way in which the grammars γ_3 – γ_6 differ from γ_1 goes beyond equivalence.

γ_1	γ_2	γ_3	γ_4	γ_5	γ_6
$x \rightarrow y$	$x \rightarrow z$	$x \rightarrow y$	$x \rightarrow z$	$x \rightarrow y$	$x \rightarrow y$
$y \rightarrow z$	$y \rightarrow z$	$y \rightarrow z'$		$y \rightarrow z$	$y \rightarrow z$
				$y \rightarrow z'$	$z \rightarrow \epsilon$

Fig. 4. Illustration of grammar relations

3.2 Beyond equivalence

Equivalence is often too restrictive to characterise related grammars. Completing a grammar, or revising a grammar, we still would like to characterise the relation between input and output grammar. We can think of various relations on two grammars γ and γ' :

- γ and γ' are only equivalent modulo renaming.
- γ' contains definitions for nonterminals neither defined nor used in γ .
- It does not hold $\llbracket n \rrbracket_{\gamma} = \llbracket n \rrbracket_{\gamma'}$, but $\llbracket n \rrbracket_{\gamma} \subseteq \llbracket n \rrbracket_{\gamma'}$ for some n .
- Some bottom nonterminals from γ are defined in γ' .

These relaxations are formalised in the following definitions.

⁵ The third condition is not implied by the first due to pathological cases for looping nonterminals. Add, for example, $z \rightarrow z$ to γ_1 from Fig. 4. The resulting grammar satisfies the first and the second condition, but not the third.

Definition 2 (Equivalence modulo renaming ρ ($\stackrel{\rho}{\equiv}$)). γ and γ' are equivalent modulo renaming ρ ($\gamma \stackrel{\rho}{\equiv} \gamma'$) if:

1. ρ is a bijective function of type $\rho : \mathcal{N}(\gamma) \rightarrow \mathcal{N}(\gamma')$,
2. $\forall n \in \mathcal{N}(\gamma). \rho(\llbracket n \rrbracket_\gamma) = \llbracket \rho(n) \rrbracket_{\gamma'}$,⁶
3. $n \in \perp(\gamma)$ implies $\rho(n) \in \perp(\gamma')$, and $n \in \mathcal{D}(\gamma)$ implies $\rho(n) \in \mathcal{D}(\gamma')$.

In Fig. 4, it holds that $\gamma_1 \stackrel{id[z'/z]}{\equiv} \gamma_3$.⁷

Definition 3 (Subgrammar relation ($\stackrel{\subseteq}{\equiv}$)). γ is a subgrammar of γ' ($\gamma \stackrel{\subseteq}{\equiv} \gamma'$) if:

1. $\mathcal{N}(\gamma) \subseteq \mathcal{N}(\gamma')$,
2. $\forall n \in \mathcal{N}(\gamma). \llbracket n \rrbracket_\gamma = \llbracket n \rrbracket_{\gamma'}$,
3. $\perp(\gamma) \subseteq \perp(\gamma')$, $\mathcal{D}(\gamma) \subseteq \mathcal{D}(\gamma')$.

In Fig. 4, it holds that $\gamma_4 \stackrel{\subseteq}{\equiv} \gamma_1$ (but of course not vice versa). As the definition details, the super-grammar γ' might employ more nonterminals than the subgrammar γ (cf. 1.). The denotations of γ are preserved (cf. 2.). Also, nonterminals of the subgrammar do not change their status to be a defined or a bottom nonterminal (cf. 3.). Thus, we can only add definitions for fresh nonterminals.

Definition 4 (Enrichment relation ($\stackrel{\supseteq}{\equiv}$)). γ' is richer than γ ($\gamma \stackrel{\supseteq}{\equiv} \gamma'$) if:

1. $\mathcal{N}(\gamma) \subseteq \mathcal{N}(\gamma')$,
2. $\forall n \in \mathcal{N}(\gamma). \llbracket n \rrbracket_\gamma \subseteq \llbracket n \rrbracket_{\gamma'}^{\perp(\gamma') \cup \emptyset(\gamma') \cup \emptyset(\gamma)}$,
3. $\perp(\gamma) \subseteq \perp(\gamma')$, $\mathcal{D}(\gamma) = \mathcal{D}(\gamma')$.

In Fig. 4, it holds that $\gamma_1 \stackrel{\supseteq}{\equiv} \gamma_5$. The above definition essentially says that γ' is richer than γ if γ' generates more than γ , i.e., $\llbracket n \rrbracket_\gamma \subseteq \llbracket n \rrbracket_{\gamma'}$. However, the actual definition also observes the looping nonterminals from γ for the denotation of γ' . Thereby, the relation is made robust regarding the particular case that γ' is obtained from γ by providing a base case for a looping nonterminal. While the subgrammar relation is concerned with the addition of definitions for fresh nonterminals, the enrichment relation is concerned with the extension of existing definitions. What remains, is a relation which covers the addition of definitions for bottom nonterminals.

Definition 5 (Instance relation ($\stackrel{\Rightarrow^*}{\equiv}$)). γ' is an instance of γ ($\gamma \stackrel{\Rightarrow^*}{\equiv} \gamma'$) if:

1. $\mathcal{N}(\gamma) \subseteq \mathcal{N}(\gamma')$,
2. $\forall n \in \mathcal{N}(\gamma). \forall x \in \llbracket n \rrbracket_\gamma. \exists y \in \llbracket n \rrbracket_{\gamma'}. x \Rightarrow_{\gamma'}^* y$,
3. $\forall n \in \mathcal{N}(\gamma). \forall y \in \llbracket n \rrbracket_{\gamma'}. \exists x \in \llbracket n \rrbracket_\gamma. x \Rightarrow_{\gamma'}^* y$,

⁶ Here, we assume that ρ cannot just be applied to nonterminals, but also to sets of strings over terminals and observable nonterminals through natural lifting.

⁷ id denotes the identity function. $id[z'/z]$ denotes the update of id at z to return z' .

4. $\mathcal{D}(\gamma) \subseteq \mathcal{D}(\gamma')$, $\mathcal{D}(\gamma') \setminus \mathcal{D}(\gamma) \subseteq \perp(\gamma)$.

In Fig. 4, it holds that $\gamma_1 \stackrel{\Rightarrow^*}{\equiv} \gamma_6$ because the bottom nonterminal z of γ_1 is resolved in γ_6 . In the definition, the second condition means that strings from $\llbracket n \rrbracket_\gamma$ can be completed into strings from $\llbracket n \rrbracket_{\gamma'}$ using $\Rightarrow_{\gamma'}^*$. The third condition states that all strings from $\llbracket n \rrbracket_{\gamma'}$ have to be reachable in this manner to make sure that the enrichment relation is not subsumed by the instance relation. A notion which can be characterised by the instance relation is context-free substitution.

3.3 Grammar transformers

The above grammar relations can be employed to define various preservation properties for grammar transformations. We model grammar transformations as partial functions on grammars, say partial *grammar transformers*.

Definition 6 (Preservation properties). *Given a partial function $f : \mathcal{G} \rightarrow \mathcal{G}$, we use $Rel(f) \subseteq \mathcal{G} \times \mathcal{G}$ to denote the relation encoded by f , that is, $Rel(f) = \{\langle \gamma, \gamma' \rangle \in \mathcal{G} \times \mathcal{G} \mid \gamma' = f(\gamma)\}$. The function f is*

1. strictly semantics-preserving if $Rel(f) \subseteq \equiv$,
2. semantics-preserving modulo renaming ρ if $Rel(f) \subseteq \stackrel{\rho}{\equiv}$,
3. introducing if $Rel(f) \subseteq \stackrel{\leq}{\equiv}$,
4. eliminating if $Rel(f) \subseteq \stackrel{\leq^{-1}}{\equiv}$,
5. increasing if $Rel(f) \subseteq \stackrel{\subset}{\equiv}$,
6. decreasing if $Rel(f) \subseteq \stackrel{\subset^{-1}}{\equiv}$,
7. resolving if $Rel(f) \subseteq \stackrel{\Rightarrow^*}{\equiv}$,
8. rejecting if $Rel(f) \subseteq \stackrel{\Rightarrow^{*-1}}{\equiv}$.

The ultimate goal is to come up with transformation operators which separate the preservation properties, and to show that the resulting operator suite for grammar adaptation is reasonably orthogonal, complete, and usable.

4 The transformation framework

We define a simple framework for grammar transformations offering transformation primitives and combinators. We give a denotational semantics of the operators using partial grammar transformers as denotations. We discuss a number of supplementary concepts such as focus, constraints and symbolic operands.

4.1 Primitives

All the operators we have in mind are derivable from primitives the syntax of which is defined in Fig. 5. There are the constant operators `id` corresponding to the identity function on grammars, `fail` modelling the undefined grammar transformation, and `reset` denoting the grammar transformer returning the empty set of rules. There are primitives to add and to subtract a rule from the given grammar. There are also primitives to substitute nonterminals, and to replace phrases by other phrases in rules. We will see later that the operator `delete` used in the introductory example is derived from the operator `replace`.

$$\begin{aligned} \textit{Trafo} &::= \textit{id} \mid \textit{fail} \mid \textit{reset} \mid \textit{add Rule} \mid \textit{sub Rule} \mid \textit{substitute Nt by Nt} \mid \textit{replace Phrase by Phrase} \\ \textit{Nt} &::= \mathcal{U}_N \\ \textit{Phrase} &::= (\mathcal{U}_N \cup \mathcal{U}_T)^* \\ \textit{Rule} &::= \textit{Nt} \rightarrow \textit{Phrase} \end{aligned}$$

Fig. 5. Syntax of the transformation primitives

In Fig. 6, the simple interpretation of the primitives is given by recursive functions in the style of denotational semantics. We use partial grammar transformers as denotations.⁸ In the definition of `substitute` and `replace`, we employ an auxiliary function `rhs` to traverse right-hand sides of rules using list-processing functions `head` and `tail`.⁹

The primitives do not satisfy any convenient preservation properties except a few trivial ones. The denotations $\mathcal{IT} \llbracket \textit{id} \rrbracket$ and $\mathcal{IT} \llbracket \textit{fail} \rrbracket$, for example, are strictly semantics-preserving. More interesting preservation properties will be enabled if we consider suitably restricted applications of the other operators. Consider, for example, the transformation `substitute n by n'` . The substitution is semantics-preserving modulo renaming if n' is fresh in the given grammar. Indeed, our operator suite for grammar adaptation will consist of such suitably restricted combinations.

4.2 Combinators

We need several combinators for grammar transformations. Firstly, there is a simple combinator $T_1;T_2$ for sequential composition of the transformations T_1 and T_2 . Secondly, there is a form of a conditional `if C then T_1 else T_2` to perform

⁸ The given semantics does not rely on `cpos` as semantic domains because general recursion is not involved. The undefined value corresponds to failure. We assume a strict failure model, that is, transformations cannot observe or recover from failure. For brevity, we do not spell out the propagation of failure.

⁹ The `rhs` traversal for `substitute` adheres to the map scheme for lists (of grammar symbols), whereas the `rhs` traversal for `substitute` relies on associative matching.

$$\begin{aligned}
\mathcal{IT} & : \text{Trafo} \rightarrow (\mathcal{G} \rightarrow \mathcal{G}) \\
\mathcal{IT} \text{ [id]} \gamma & = \gamma \\
\mathcal{IT} \text{ [fail]} \gamma & = \text{undefined} \\
\mathcal{IT} \text{ [reset]} \gamma & = \emptyset \\
\mathcal{IT} \text{ [add } r] \gamma & = \gamma \cup \{r\} \\
\mathcal{IT} \text{ [sub } r] \gamma & = \gamma \setminus \{r\} \\
\mathcal{IT} \text{ [substitute } n \text{ by } n'] \gamma & = \{ren(n) \rightarrow rhs(u) \mid n \rightarrow u \in \gamma\} \\
& \text{where} \\
ren(m) & = id[n'/n](m) \\
rhs(u) & = \begin{cases} \epsilon, & \text{if } u = \epsilon \\ ren(head(u)) \ rhs(tail(u)), & \text{otherwise} \end{cases} \\
\mathcal{IT} \text{ [replace } p \text{ by } p'] \gamma & = \{n \rightarrow rhs(u) \mid n \rightarrow u \in \gamma\} \\
& \text{where} \\
rhs(u) & = \begin{cases} \text{undefined}, & \text{if } p = \epsilon \\ \epsilon, & \text{if } u = \epsilon \\ p' \ rhs(v), & \text{if } u = p \ v \\ head(u) \ rhs(tail(u)), & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 6. Semantics of the transformation primitives

either the transformation T_1 or the transformation T_2 depending on the condition C . We postpone discussing possible forms of conditions. Thirdly, there is a combinator T/ψ to apply a transformation T in a focus, that is, for certain nonterminals ψ only. Recall the introductory example: Some phrase had to be deleted in the definition of REDEFINES-clause (but not elsewhere). Finally, there is a trivial combinator $T!$ to enforce that a given transformation has an effect. An adaptation step, which has no effect, usually indicates an error. In Fig. 7, the syntax for the combinators is summarised.

$$\begin{aligned}
\text{Trafo} & ::= \dots \mid \text{Trafo}; \text{Trafo} \mid \text{if } Cond \text{ then } \text{Trafo} \text{ else } \text{Trafo} \mid \text{Trafo}/Focus \mid \text{Trafo}! \\
Focus & ::= \{Nt, \dots, Nt\}
\end{aligned}$$

Fig. 7. Syntax of the transformation combinators

The semantics of the combinators is defined in Fig. 8. The interpretation of sequential composition and of the if-construct is straightforward. In the equation for a focused transformation T/ψ , we use an auxiliary operator γ/ψ to restrict a grammar γ to a set of nonterminals ψ . The operator is defined as follows:

$$\gamma/\psi = \{n \rightarrow u \in \gamma \mid n \in \psi\}$$

The interpretation of T/ψ details that the remainder of the grammar outside of the focus, that is, $\gamma/\mathcal{D}(\gamma) \setminus \psi$, is preserved. Note also that an invalid focus (i.e., $\psi \not\subseteq \mathcal{D}(\gamma)$) leads to an undefined result. The equation for the interpretation of $T!$ formalises what we mean by a transformation that has an effect on the given

grammar. The transformation $T!$ only succeeds if T succeeds *and* the resulting grammar is different from the given grammar. Here, we assume structural equality on grammars. Let us consider two examples. The transformation $\text{id}!$ unconditionally fails. The transformation substitute n by $n'!$ fails if there are no occurrences of n in the given grammar, or if $n = n'$.

$$\begin{aligned}
\mathcal{IT} & : \text{Trafo} \rightarrow (\mathcal{G} \leftrightarrow \mathcal{G}) \\
\mathcal{IT} \llbracket T_1; T_2 \rrbracket \gamma & = \mathcal{IT} \llbracket T_2 \rrbracket (\mathcal{IT} \llbracket T_1 \rrbracket \gamma) \\
\mathcal{IT} \llbracket \text{if } C \text{ then } T_1 \text{ else } T_2 \rrbracket \gamma & = \begin{cases} \mathcal{IT} \llbracket T_1 \rrbracket \gamma, & \text{if } c = \text{true} \\ \mathcal{IT} \llbracket T_2 \rrbracket \gamma, & \text{if } c = \text{false} \end{cases} \\
& \quad \text{where } c = \mathcal{IC} \llbracket C \rrbracket \gamma \\
\mathcal{IT} \llbracket T/\psi \rrbracket \gamma & = \begin{cases} \gamma', & \text{if } \psi \subseteq \mathcal{D}(\gamma) \\ \text{undefined,} & \text{otherwise} \end{cases} \\
& \quad \text{where } \gamma' = (\mathcal{IT} \llbracket T \rrbracket \gamma/\psi) \cup \gamma/\mathcal{D}(\gamma)\setminus\psi \\
\mathcal{IT} \llbracket T! \rrbracket \gamma & = \begin{cases} \gamma', & \text{if } \gamma \neq \gamma' \\ \text{undefined,} & \text{otherwise} \end{cases} \\
& \quad \text{where } \gamma' = \mathcal{IT} \llbracket T \rrbracket \gamma
\end{aligned}$$

Fig. 8. Semantics of the transformation combinators

4.3 Constraints

In order to derive operators which satisfy some interesting preservation properties, we need means to constrain transformations. One constraint which we already encountered is that a transformation has to be effective. Other constraints can be formulated using conditionals (if $C \dots$). In Fig. 9, some forms of conditions C are defined. In practice, further forms might be relevant. There are conditions which are concerned with the kinds of nonterminals (*fresh*, *bottom*, etc.). Such conditions are valuable if we need to enforce assumptions on the nonterminals occurring as operands in transformations. There is a further form of condition, namely u covers v . Roughly, the condition attempts to test if u generates at least what v generates. Such tests are useful, for example, to constrain increasing and decreasing transformations.

The evaluation of the conditions concerned with the various kinds of nonterminals simply refers to the corresponding sets of nonterminals defined earlier. In the evaluation of u covers v , we use an auxiliary binary relation \rightsquigarrow_γ . Conceptually, $u \rightsquigarrow_\gamma v$ should hold if the observable strings derivable from v are also derivable from u (but not necessarily vice versa). In practice, we use a pessimistic heuristic as indicated in Fig. 9 to check for coverage. Such a heuristic checks if

Syntax

$Cond ::= \text{fresh } Nt \mid \text{bottom } Nt \mid \text{top } Nt \mid \text{used } Nt \mid \text{defined } Nt \mid \text{looping } Nt$
 $\mid Nt \in Focus \mid Phrase \text{ covers } Phrase$

Evaluation

$\mathcal{IC} : Cond \rightarrow (\mathcal{G} \rightarrow \{false, true\})$

$\mathcal{IC} \llbracket \text{fresh } n \rrbracket \gamma = n \notin \mathcal{N}(\gamma)$

$\mathcal{IC} \llbracket \text{bottom } n \rrbracket \gamma = n \in \perp(\gamma)$

...

$\mathcal{IC} \llbracket n \in \psi \rrbracket \gamma = n \in \psi$

$\mathcal{IC} \llbracket u \text{ covers } v \rrbracket \gamma = u \rightsquigarrow_{\gamma} v$

Coverage

$u \rightsquigarrow_{\gamma} u$

$n \rightsquigarrow_{\gamma} u$ where $n \rightarrow u \in \gamma$

$u \rightsquigarrow_{\gamma} n$ where $\gamma/\{n\} = \{n \rightarrow u\}, n \notin \emptyset(\gamma)$

...

Abbreviations

$C?$ = if C then id else fail

$\neg C?$ = if C then fail else id

$u \text{ equiv } v?$ = $u \text{ covers } v?$; $v \text{ covers } u?$

Fig. 9. Syntax and evaluation of conditions

the two given phrases can be made (structurally) equal by a finite number of symbolic derivation steps in the sense of \Rightarrow_{γ} .¹⁰

We also define some convenient abbreviations in Fig. 9 for transformations which serve solely as guards. The transformation $C?$ models a guard with a positive condition C , whereas $\neg C?$ models a guard with a negated condition C . We will use such guards to model pre- and post-conditions of transformation operators. The last abbreviation is a guard $u \text{ equiv } v?$ for checking two phrases to be equivalent by a “conjunction” checking coverage in both directions.

Finally, we should comment on the interaction of the focus-construct and the evaluation of conditions. Since conditions are evaluated w.r.t. a given grammar, it matters if a guard is performed before or after restriction, and before or after re-unification with the grammar outside a focus. Usually, one wants to place conditionals before restriction or after re-unification. We cannot check, for example, that a nonterminal is fresh by just looking at a restricted grammar. One can think of other more flexible semantic models for the focus-construct, e.g., a model where the restriction is not performed by the focus-construct itself, but rather by the unconditional primitives in the last possible moment. Such a model has the drawback that the propagated focus has to be kept consistent. This is not straightforward.

¹⁰ Of course, we cannot determine in general if the strings generated by v are also generated by u because otherwise we would claim that the subset relationship for context-free grammars is decidable.

4.4 Symbolic operands

So far, transformations only operate on *concrete* nonterminals, phrases, rules, focuses as operands due to the definition of *Nt*, *Phrase*, *Rule*, and *Focus* (cf. Fig. 5 and Fig. 7). This is not always convenient. Sometimes, we also would like to formulate *symbolic* operands for transformations.

```

Syntax
Phrase ::= ... | definition of Nt
Focus  ::= ... | all

Normalisation
definition of n →γ u where γ/{n} = {n → u}
all          →γ D(γ)

```

Fig. 10. Two forms of symbolic operands

Let us consider examples. A symbolic form of phrase is *definition of n* which denotes the right-hand side of the rule defining n assuming there is only a single rule with n on the left-hand side. This form is convenient, for example, in the context of unfolding, where a nonterminal is replaced (in terms of the primitive replace) by the right-hand side of the definition of the nonterminal. We are relieved from actually pointing out the definition itself. Another symbolic form is *all*. It denotes all nonterminals defined in the grammar at hand. This form is useful if an operator expecting a focus parameter should be applied globally. We are relieved from enumerating all defined nonterminals. In practice, other forms are relevant, too. We can think of, for example, a form to turn a phrase into the corresponding permutation phrase [4]. If extended BNF is taken into account, all forms for alternatives, optionals, lists also need to be dealt with symbolically.

The question is how symbolic forms should be evaluated. Recall that conditionals are evaluated by the semantic function \mathcal{IC} which in turn is invoked if conditionals are interpreted via \mathcal{IT} . We could adopt this approach for symbolic forms, and introduce corresponding evaluation functions for *Nt*, *Phrase*, *Rule* and *Focus* which are then invoked during interpretation of the transformations. For pre- and post-conditions it is sensible, that they are evaluated w.r.t. intermediate grammars in a compound transformation. It is not sensible for symbolic forms. Consider, for example, a transformation for unfolding the definition of n in the focus of $\{n'\}$. The definition of n is not available for evaluation after the restriction to the focus $\{n'\}$. We propose that symbolic forms of a given transformation corresponding to an *indivisible adaptation step* are eliminated before interpretation. This model provides referential transparency. In Fig. 10, elimination rules for the above examples of symbolic forms are shown.¹¹

¹¹ We conceive these elimination rules (performed w.r.t. a grammar γ) as rewrite rules to be used for normalisation. If the resulting normal form still contains symbolic forms, we consider that as an error, and we will not interpret the corresponding transformation.

5 The operator suite

We are now in the position to derive operators which are immediately useful for actual grammar adaptation. Indeed, the derived operators are meant to mimic the roles one naturally encounters while adapting grammars manually.

5.1 Overview

A *stepwise adaptation* can be conceived as a sequence $T_1; \dots; T_m$ of transformation steps, where the T_i correspond to applications of the operators from the suite. A transformational grammar programmer does neither use primitives nor combinators. He solely uses operators of the suite. In particular, a programmer does not restrict the focus of transformations himself. Instead, operators expose a focus parameter if necessary. The T_i are exactly those indivisible adaptation steps assumed for the evaluation model of symbolic operands.

Refactoring

preserve P in F as P'	$= P \text{ equiv } P' ?$; replace P by $P' !_{/F}$; $P \text{ equiv } P'$
fold P in F to N	$= \text{introduce } N \text{ as } P$; preserve P in F as $N ?$
unfold N in F	$= \text{preserve } N \text{ in } F \text{ as definition of } N$
introduce N as P	$= \text{fresh } N ?$; add $N \rightarrow P$
eliminate N	$= \text{reject } \{N\}$; fresh $N ?$
rename N to N'	$= \text{fresh } N' ?$; substitute N by $N' !$

Construction

generalise P in F to P'	$= P' \text{ covers } P ?$; replace P by $P' !_{/F}$; $P' \text{ covers } P ?$
include P for N	$= \text{defined } N ?$; add $N \rightarrow P !$
resolve N as P	$= \text{bottom } N ?$; add $N \rightarrow P$
unify N to N'	$= \text{bottom } N ?$; $\neg \text{fresh } N' ?$; replace N by N'

Destruction

restrict P in F to P'	$= P \text{ covers } P' ?$; replace P by $P' !_{/F}$; $P \text{ covers } P' ?$
exclude P from N	$= \text{sub } N \rightarrow P !$; defined $N ?$
reject F	$= \text{reset}_{/F}$
separate N in F as N'	$= \text{fresh } N' ?$; replace N by $N' !_{/F}$; $\neg \text{fresh } N ?$
delete P in F	$= \neg P \text{ covers } \epsilon ?$; replace P by $\epsilon !_{/F}$

Fig. 11. Operator suite for grammar adaptation

The definitions of all the operators covered in this section are shown in Fig. 11. The operators can be subdivided into three groups, namely refactoring, construction, and destruction. These groups are discussed in detail below. Note that several of these operators are more interesting if extended BNF expressiveness was considered. Properties of the operators, mainly semantics preservation properties, will be summarised at the end of the section.

5.2 Refactoring

Refactoring is useful to restructure grammars so that comprehensibility is improved, or subsequent adaptation steps are easier to perform. Refactoring operators are semantics-preserving in the narrow sense. Note how the guards in the operator definitions describe pre- and post-conditions. Also note that the operators check themselves if they actually affect the given grammar (either directly by “!”, or more indirectly).

A fundamental refactoring operator is `preserve`. The operator allows to replace a phrase by an equivalent one. There is a focus parameter (cf. `... in ...`), i.e., one can restrict the replacement to a certain focus. The constraint that the old and the new phrase have to be equivalent is checked before and after replacement.¹² Operators for folding and unfolding nonterminal definitions can be directly derived from the operator `preserve`. Folding and unfolding are common transformations discussed for various formalisms elsewhere (cf. [12]). Nonterminals which should be unfolded have to be defined by exactly one rule.¹³ The operator `preserve` is not restricted to folding and unfolding. In general, it can be used, if a grammar needs to be simplified, e.g., to remove an ambiguity, or if a grammar should be restructured to adhere to a certain style, e.g., for turning recursive nonterminals into extended BNF notation for lists.

Let us consider the remaining refactoring operators. Introduction and elimination of nonterminals is facilitated by the operators `introduce` and `eliminate`. Introduction can be clearly conceived as semantics-preserving in the narrow sense because all previous definitions are preserved. We consider elimination also as semantics-preserving in the narrow sense because we can only eliminate nonterminal definitions which are not needed in the remaining grammar anyway. Here, we assume that a grammar programmer has a clear perception of what the primary nonterminals of a grammar are, e.g., in the sense of a start symbol, and that such primary nonterminals are not eliminated. Finally, there is the operator `rename` for renaming of nonterminal symbols. The pre-condition `fresh N'` for `rename N to N'` makes sure that `N` is not renamed to a nonterminal `N'` which is already in use in the given grammar. This requirement qualifies substitution to perform renaming.

There is an insightful asymmetry between folding and unfolding. The operator `fold` performs introduction on the fly, whereas the operator `unfold` does not perform elimination. This asymmetry is sensible for pragmatic reasons. Unfolding is often performed in a focus and not globally. Thus, elimination is usually impossible. By contrast, folding is usually performed to identify a subphrase and to introduce a nonterminal for it. Otherwise, the operator `preserve` can be used.

¹² The post-condition is needed to cope with pathological cases. Consider, for example, `preserve` definition of `n` in `{n}` as `n`, i.e., a transformation where the definition of a nonterminal `n` is replaced by `n` itself. The definition of `n` would be damaged by this replacement.

¹³ This is implied by the use of definition of `... in` in the definition of `unfold`.

5.3 Construction

The second group of operators in Fig. 11 facilitates construction. The term construction is meant here in a rather broad sense, that is, grammar substitution, extension, completion and others. Let us discuss the various operators in detail.

generalise P in F to P' The phrase P is replaced by the more general phrase P' in all occurrences focused by F . This is one convenient way to extend the generated language. As the pre-condition details, P' has to cover P .

include P for N Another straightforward way to extend the generated language is to add a rule $N \rightarrow P$ for a nonterminal N already defined in the given fragment. In fact, the pre-condition makes sure that there is a definition to be extended. Compared to the operator **generalise**, the operator **include** works at the rule level and not inside rules.

resolve N as P This operator is used to provide the definition for a bottom nonterminal N . Note that both **include** and **resolve** add rules, but due to the guards, the roles of increasing and resolving transformations are separated.

unify N to N' The bottom nonterminal N is unified with another nonterminal N' in the sense that N is replaced by N' . The operator **unify** is useful, if a bottom nonterminal should be resolved in terms of an existing definition (cf. first scenario in Fig. 12), or if two bottom nonterminals intentionally coincide (cf. second scenario in Fig. 12).

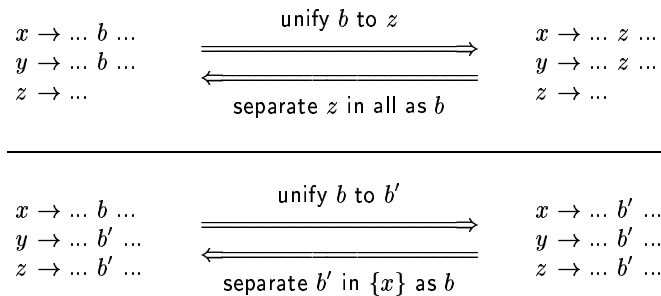


Fig. 12. Illustration of the operators **unify** and **separate**

The operators for construction are useful in grammar development to complete and to connect a grammar. Missing rules are added by **include**. Too restrictive phrases are generalised with **generalise**. Missing definitions of nonterminals are established via **resolve**. Nonterminals are unified with **unify**.

5.4 Destruction

The third group of operators in Fig. 11 is useful for destructing grammars. Again, destruction is meant here in a broad sense. The destruction group provides essentially the inverse operators of the construction group.¹⁴ The operator

¹⁴ Exception: The operator **delete** has no counterpart in the construction group.

`restrict` replaces a phrase by a more restrictive phrase in a given focus. The operator `exclude` excludes one rule for a given nonterminal. The post-condition makes sure that `exclude` behaves as a decreasing rather than a rejecting operator. The latter role is solely modelled by the operator `reject`, and accordingly enforced by its post-condition. The operator `separate` “breaks up connections” in the grammar, that is, it replaces nonterminal occurrences by a fresh nonterminal as illustrated in Fig. 12. The post-condition ensures that the separation does not degenerate to a renaming. Finally, the operator `delete` removes a phrase from the focused rules. It has been illustrated in the introduction. The pre-condition of `delete` is insightful because it makes sure that this removal cannot be done in a more modest way, that is, by restricting the phrase of concern to ϵ via the more disciplined operator `restrict`.

As the operators for construction are useful in grammar development to complete and to connect a grammar, the destruction operators are useful for correction or revision. Too general phrases can be restricted. Superfluous rules or definitions can be excluded or rejected, respectively. Accidentally unified phrases can be separated by introducing new nonterminals in certain occurrences. Refactoring steps usually precede construction and destruction.

5.5 Discussion

Let us discuss the properties of the various operators. The semantics-preservation properties are summarized in Fig. 13. The figure also shows the (sometimes only approximative) inverse operator for each operator. The operators from the refactoring group are semantics-preserving in the narrow sense, say strictly semantics-preserving, semantics-preserving modulo renaming, introducing, or eliminating. The operators for construction and destruction are not semantics-preserving in the narrow sense. Several operators experience the grammar relations for enrichment and instances, that is, the corresponding transformations are increasing, decreasing, resolving or rejecting.

The properties from Fig. 13 are implied by simple arguments. Let us consider a few examples. The operator `preserve` is strictly preserving because equivalence of the phrases (w.r.t. generation of observable strings) involved in replacement implies equivalence of grammars. Since unfolding is solely defined in terms of `preserve`, strict semantics-preservation for `unfold` is implied. The operator `introduce` is introducing because it adds a rule for a fresh nonterminal. This is a direct implementation of the notion of a super-grammar. The operator `fold` is introducing because it is defined as a sequence of an introducing transformation and a strictly semantics-preserving transformation. A rigorous proof for the properties is beyond the scope of the paper.

The distribution of the various preservation properties in Fig. 13 documents a kind of orthogonality of the operator suite. Most operators experience exactly one kind of preservation property in a pure way. All operators except `unify`, `separate`, and `delete` are pure in that sense. All grammar relations or preservation properties resp. are covered by the pure operators. The impure operators have been introduced for convenience. In practice, a few further impure operators are

Operator	Semantics Preservation	Inverse
Refactoring		
preserve	strict	preserve
fold	introducing	unfold
unfold	strict	fold
introduce	introducing	eliminate
eliminate	eliminating	introduce
rename	modulo renaming	rename
Construction		
generalise	increasing	restrict
include	increasing	exclude
resolve	resolving	reject
unify	essentially resolving	separate
Destruction		
restrict	decreasing	generalise
exclude	decreasing	include
reject	rejecting	resolve
separate	essentially rejecting	unify
delete	“zig-zag”	

Fig. 13. Properties of the derived operators

convenient, too. In fact, the impure operators can be reconstructed in terms of the pure operators. These reconstructions are also useful to understand the exact preservation properties of the impure operators. We only show the reconstruction of `unify`:¹⁵

`unify N to N' = ¬ fresh N' ?; resolve N by N'; unfold N in all; eliminate N`

Superficially, `unify` is a resolving operator since it “connects” one nonterminal N to another N' , that is, N is defined in terms of N' . As we see, the transformation sequence above is indeed essentially a resolving transformation. However, the nonterminal N is only defined temporarily. The definition is eliminated after unfolding. The complete transformation sequence is not strictly resolving because the instance relationship does not hold without the temporary definition.

6 Concluding remarks

Towards proper grammar engineering As for common practice, grammar development is usually done in ad-hoc manner. Grammar development is often conceived as coding effort, as grammar (conflict) hacking. As for research, grammars are not too much of a topic anymore. Grammars deserve more attention

¹⁵ Reconstruction of `separate` and `delete` in terms of the pure operators is slightly more tedious since we need fresh nonterminals for some temporary definitions. This is also the reason why we favoured a more compact (impure) definition solely based on `replace` in Fig. 11.

both in practice and academia. Grammars should be regarded much more as real engineering artifacts. Research should supply the methods useful in grammar development. The present paper contributes in this context. The paper lays down the foundations of an adaptive style for grammar development. Grammars are adapted by well-behaved program transformations. The ultimate grammar programmer probably does not write down verbose transformations, but he rather uses an interactive tool which automatically deduces grammar transformations. One can think of a proper grammar engineering environment providing not just support for interactive adaptation but also for grammar assessment, testing by parsing, coverage measurement, and others. Similar tools are envisioned and described (to some extent) in [14, 9].

Semantics-preservation and relaxation To call one group of adaptation operators refactorings, has been inspired by the idea of refactoring in (object-oriented) programming [10]. Indeed, the intentions coincide, that is, programs or grammars resp. are adapted in a semantics-preserving manner to improve their structure, and to prepare subsequent extensions or even revisions. Recall that the other groups of adaptation operators discussed in the paper are not semantics-preserving in the narrow sense. The major body of research on transformational programming assumes semantics-preservation [11, 12]. The present paper contributes a set of weaker preservation notions which are suitable to characterise revisions, extensions and others. The style of adaptive grammar programming developed in the paper scales up to practically relevant grammar projects [9], and it helps to actually make these projects feasible and predictable.

Perspective The present paper does not explore several dimensions in depth. The operator suite is complete in a trivial sense: One can derive any grammar with it.¹⁶ We are interested in more global notions than preservation properties, e.g., if a grammar is *improved* in some sense along some transformation sequence. This is a challenging research topic. Another issue ignored in the present paper is the interaction of context-free syntax and semantic functions as relevant for attribute grammars. This issue is in fact practically relevant for the application of compiler compilers like YACC. At a more general level, this interaction issue can be rephrased as the question how clients of a grammar such as compiler compiler inputs, rewrite rules have to be adapted if the grammar serving as contract changes. This issue is an open research problem. Finally, we would like to integrate our approach with other scenarios of grammar adaptation, and applications for grammar transformations. We just mention a few. In [16], grammar transformations are used for the development of domain-specific languages starting from reusable syntax components. In [15], grammar transformations are used to derive abstract syntaxes from concrete syntaxes. In several contexts, schematic adaptation is often relevant as opposed to stepwise adaptation favoured in the present paper. For schematic adaptation, one needs to formulate transformation

¹⁶ An extreme strategy is the following. The input grammar is discarded by reject all. The output grammar is constructed (from scratch) via introduce, resolve, and include.

schemes which are then systematically applied all over the grammar either exhaustively or according to some heuristics. In grammar reengineering [14], one is, for example, concerned with DeYACCification, that is, the systematic introduction of optionals, lists and that alike. In parser implementation [2, 1, 13], schematic adaptation is needed to systematically eliminate list constructs, to optimise grammars w.r.t. certain grammar classes, or for grammar class migration.

References

1. J. Aycock and N. Horspool. Faster Generalized LR Parsing. In S. Jähnichen, editor, *Proc. of the 8th International Conference on Compiler Construction (CC'99)*, volume 1575 of *LNCS*, pages 32–46. Springer-Verlag, 1999.
2. D. Blasband. *Automatic Analysis of Ancient Languages*. PhD thesis, Free University of Brussels, 2000.
3. M. Brand, M. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming*, 36(2–3):209–266, 2000.
4. R. D. Cameron. Extending context-free grammars with permutation phrases. *ACM Letters on Programming Languages and Systems*, 2(4):85–94, Mar. 1993.
5. E. Chikofsky and J. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.
6. M. de Jonge and J. Visser. Grammars as Contracts. In *Proc. of GCSE 2000*, LNCS, Erfurt, Germany, 2001. Springer-Verlag. To appear.
7. IBM Corporation. *VS COBOL II Application Programming Language Reference*, 1993. Release 4, Document number GC26-4047-07.
8. R. Lämmel and C. Verhoef. VS COBOL II Grammar Version 1.0.3. <http://www.cs.vu.nl/grammars/browsable>, 1999–2001.
9. R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. Submitted, available at <http://www.cwi.nl/~ralf/>, July 2000.
10. W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
11. H. A. Partsch. *Specification and Transformation of Programs*. Springer-Verlag, 1990.
12. A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, June 1996.
13. J. J. Sarbo. Grammar transformations for optimizing backtrack parsers. *Computer Languages*, 20(2):89–100, May 1994.
14. M. Sellink and C. Verhoef. Development, Assessment, and Reengineering of Language Descriptions. In J. Ebert and C. Verhoef, editors, *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pages 151–160. IEEE Computer Society, March 2000.
15. D. S. Wile. Abstract Syntax From Concrete Syntax. In *Proc. of the 1997 International Conference on Software Engineering*, pages 472–480. ACM Press, 1997.
16. D. S. Wile. Integrating Syntaxes and their Associated Semantics. Draft, 1999.