Reconstruction of paradigm shifts

Ralf Lämmel[1] and Günter Riedewald[2]

1: Universität Rostock, Fachbereich Informatik
18051 Rostock, Germany
`rlaemmel@informatik.uni-rostock.de`
2: Universität Rostock, Fachbereich Informatik
18051 Rostock, Germany
`gri@informatik.uni-rostock.de`

Abstract

There are many extensions of the basic attribute grammar formalism intended to improve its pragmatics, e.g. certain modularity concepts, remote access, object-orientation, templates, rule models and higher-order features. In the paper, *a generic and formal approach to an effective and orthogonal reconstruction* of the concepts underlying some extensions is described. The reconstruction is *effective* in the sense that the reconstructed concepts are presented as executable meta-programs. The approach to reconstruction is *formal* in the sense that the derived meta-programs modelling certain concepts can be analysed based on properties of the meta-programs, e.g. preservation properties. It is a *generic* approach because the meta-programming framework can be instantiated not only for attribute grammars but also for several other representatives of the declarative paradigm, e.g. natural semantics and algebraic specification. Thereby, concepts can be imported from and exported to other frameworks. Finally, the reconstructions are derived *orthogonally* in the sense that potential roles are first unbundled and then particular combinations of the roles can be investigated. The described meta-programming framework has been implemented in the specification framework of $\Lambda\Delta\Lambda$ and it is used for reusable formal language definition based on attribute grammars and operational semantics.

1. Introduction

It is generally agreed that the pragmatics of the basic attribute grammar (AG) formalism is not very suitable for practical applications. There has been a lot of research on augmenting the basic paradigm with extensions[1] to overcome certain problems such as the lack of modularity, extensibility and reusability; refer e.g. to [Wat75, Kas76, Lor77, Gie88, Hed89, DC90, Paa91, Kos91, SV91, Hed91, Kas91, FMY92, KLMM93, LJPR93, KW94, Boy96, DPRJ96, ZKM97, DPRJ97, Boy98, MZ98]. Several concepts such as remote access, object-orientation, templates, rule models, symbol computations, etc. have been implemented in one or another AG specification language such as *Lido* (*Eli*) [Kas91, KW94], *Olga* (*FNC-2*) [JP91] and *Lisa* [ZKM97, MZ98]. It is an open question what combination of concepts is most appropriate and what further improvements of the pragmatics can be expected. Indeed, there is an ongoing research on further concepts, e.g. genericity [DPRJ96, DPRJ97, CDPR98], adaptive programming [CDPR98] and aspect-oriented programming [Läm99a]. Since their are so many concepts, which have been introduced in certain AG frameworks and which have been formulated and implemented in very different ways, an analysis and a comparison of them is very delicate.

[1] [KW94] suggests the term *paradigm shifts* for several extensions such as symbol computations, remote access and inheritance. Thus, the title of our paper is a homage to [KW94].

We want to approach to the reconstruction of several concepts using functional meta-programs on AG specifications as the primary tool. Essentially, several roles present in some concepts are isolated and represented as meta-programs in our framework. Investigating combinations of roles we might be able to simulate several existing concepts or we even may find unique combinations. Our approach is formal in the sense, that we use a "typeful" approach to meta-programming and properties of program transformations corresponding to the roles and combinations of them can be analysed.

[axiom]	*program*	::=	*declaration_part statement_part*
[dp]	*declaration_part*	::=	*declarations*
[sp]	*statement_part*	::=	*statements*
[decs]	*declarations*	::=	*declaration declarations*
[nodec]	*declarations*	::=	
[dec]	*declaration*	::=	*vdec type*
[concat]	*statements*	::=	*statement statements*
[skip]	*statements*	::=	
[assign]	*statement*	::=	*vuse expression*
[if]	*statement*	::=	*expression statements statements*
...			
[var]	*expression*	::=	*vuse*
[const]	*expression*	::=	*constant*
...			

Figure 1: CFG for a simple imperative language

The running example of the paper is introduced in the sequel. Consider the context-free grammar (CFG) of a simple imperative language in Figure 1. Let us assume that we are interested in all variables which are declared but which are not used in the statement part. The result of the corresponding analysis should be modelled by a synthesized attribute USELESS↑[2] of the root symbol *program*. We want to investigate several approaches to the specification of the accumulation of the variable identifiers from the declaration part and the statement part.

1. The "pure" approach to a corresponding AG specification is shown in Figure 2, where several attributes named VDECS↑ and VUSES↑ are used to perform the accumulation of all the variable occurrences all over the grammar in declarations and statements respectively. Multiple attributes in a rule are combined in the sense of the union on sets (see $\_\cup\_$ in rules [decs], [concat], [assign] and [if]), whereas single occurrences of variables are coerced to sets by the singleton set construction (see $\{\_\}$ in rules [dec], [assign] and [var]). If there are no occurrences the empty set is propagated upward the tree (see $\emptyset$ in rules [nodec], [skip] and [const]).
2. Another more modular approach is shown in Figure 3. It is based on the *Constituents*-construct for remote access [KW94] as available in the specification language *Lido* in *Eli* [Kas91]. Thereby, the remote attributes corresponding to the declared and the used attributes can directly be combined. The approach is more modular in the sense that it abstracts from the underlying CFG and there is no need to clutter the AG specification with the accumulation of variable occurrences.
3. The way we explained the first approach indicates that all the computations (semantic rules) follow a certain schema. Indeed, in the paper we will use program transformations to add the computations accordingly. The transformational approach culminates in Subsection 4.2, where a certain combination of simple transformations will be presented as a reconstruction of the *Constituents*-construct mentioned in the second approach.

[2]We adhere to the convention that names of attributes end with ↓ or ↑ corresponding to names of inherited resp. synthesized attributes. Note that the arrows are not just annotations but they are proper components of the names.

[axiom]	*program*	$::=$	*declaration_part statement_part*
	program.USELESS↑	$:=$	*declaration_part*.VDECS↑ $\setminus$ *statement_part*.VUSES↑
[dp]	*declaration_part*	$::=$	*declarations*
	declaration_part.VDECS↑	$:=$	*declarations*.VDECS↑
[sp]	*statement_part*	$::=$	*statements*
	statement_part.VUSES↑	$:=$	*statements*.VUSES↑
[decs]	*declarations*	$::=$	*declaration declarations*
	declarations.VDECS↑	$:=$	*declaration*.VDECS↑ $\cup$ *declarations*.VDECS↑
[nodec]	*declarations*	$::=$	
	declarations.VDECS↑	$:=$	$\emptyset$
[dec]	*declaration*	$::=$	*vdec type*
	declaration.VDECS↑	$:=$	{*vdec*.ID↑}
[concat]	*statements*	$::=$	*statement statements*
	statements.VUSES↑	$:=$	*statement*.VUSES↑ $\cup$ *statements*.VUSES↑
[skip]	*statements*	$::=$	
	statements.VUSES↑	$:=$	$\emptyset$
[assign]	*statement*	$::=$	*vuse expression*
	statement.VUSES↑	$:=$	{*vuse*.ID↑} $\cup$ *expression*.VUSES↑
[if]	*statement*	$::=$	*expression* $statements_1$ $statements_2$
	statement.VUSES↑	$:=$	*expression*.VUSES↑
		$\cup$	$statements_1$.VUSES↑ $\cup$ $statements_2$.VUSES↑
...			
[var]	*expression*	$::=$	*vuse*
	expression.VUSES↑	$:=$	{*vuse*.ID↑}
[const]	*expression*	$::=$	*constant*
	expression.VUSES↑	$:=$	$\emptyset$
...			

Figure 2: Detecting superfluous variable declarations — The pure approach

[axiom]	*program*	$::=$	*declaration_part statement_part*
	program.USELESS↑	$:=$	*declaration_part*.VDECS↑ $\setminus$ *statement_part*.VUSES↑
[dp]	*declaration_part*	$::=$	*declarations*
	declaration_part.VDECS↑	$:=$	**Constituents** *vdec*.ID **With** (VDECS, _ $\cup$ _, {_}, $\emptyset$)
[sp]	*statement_part*	$::=$	*statements*
	statement_part.VUSES↑	$:=$	**Constituents** *vuse*.ID **With** (VUSES, _ $\cup$ _, {_}, $\emptyset$)

Figure 3: A variant of Figure 2 using remote access

The remaining paper is structured as follows. In Section 2 a framework for transformational programming on AGs is developed. Afterwards, certain roles of program synthesis are introduced and analysed in Section 3. The corresponding operators are used in Section 4 for the reconstruction of some paradigm shifts and for the derivation of a unique notion, that is to say the schematic adaptation of computations. Finally, we comment on results, related work and future work in Section 5.

2. A framework for functional meta-programs

Our approach to the reconstruction of paradigm shifts is based on a framework for functional meta-programs; refer to [Läm99b] for details. The framework is developed in the following steps. First, a representation for declarative target programs such as attribute grammars, natural semantics, logic programs and constructive algebraic specifications is declared. Second, the corresponding structural definitions are restricted to obtain the proper domains of target program fragments. These domains are embedded into a typed λ-calculus, where some further specification constructs are added as well. Finally, properties of meta-programs, e.g. preservation properties, are defined.

2.1. The representation of target programs

The representation of declarative target programs and fragments of them is given by certain domains which are intended to capture constructs such as rules and parameters common to (first-order) declarative languages. The structural definition in Figure 4 is already refined for AGs. Note that barred names are used to point out that the structural definition needs to be restricted further to define the corresponding domain of *proper* fragments discussed in the next subsection.

$\overline{\mathsf{Rules}}$	$=$	$\overline{\mathsf{Rule}}^{\star}$	*compatible* sequences of rules
$\overline{\mathsf{Rule}}$	$=$	$\mathsf{Tag} \times \overline{\mathsf{Conclusion}} \times \overline{\mathsf{Premise}}^{\star}$	tagged rules
$\overline{\mathsf{Conclusion}}$	$=$	$\overline{\mathsf{Element}}$	conclusions ("LHSs")
$\overline{\mathsf{Premise}}$	$=$	$\overline{\mathsf{Element}} + \overline{\mathsf{Computation}}$	premises ("RHS" elements)
$\overline{\mathsf{Element}}$	$=$	$\mathsf{Name} \times \overline{\mathsf{Parameter}}^{\star} \times \overline{\mathsf{Parameter}}^{\star}$	parameterized grammar symbols
$\overline{\mathsf{Computation}}$	$=$	$\mathsf{Operator} \times \overline{\mathsf{Parameter}}^{\star} \times \overline{\mathsf{Parameter}}^{\star}$	computations (semantic rules)
$\mathsf{Parameter}$	$=$	$\mathsf{Variable} \times \mathsf{Sort}$	annotated parameters such as variables
Tag			tags
Name			grammar symbols
$\mathsf{Operator}$			semantic function symbols
$\mathsf{Variable}$			countable set of variable identifiers
Sort			countable set of sort identifiers

Figure 4: Structural definition of representations

Without going into detail it should be explained how AG specifications can be represented according to the domains in Figure 4:

- The domain Rule can be regarded as an abstraction from production rules together with the attributes + semantic rules. A rule $r \in \mathsf{Rule}$ is a triple consisting of a tag t, a conclusion $c \in \mathsf{Conclusion}$ and a sequence of premises $p^\star \in \mathsf{Premise}^\star$. The notation $[t]\ c : p^\star$ is used to construct a rule.
- The conclusion corresponds to the nonterminal on the LHS of the production rule together with variables for its attributes. Similarly, premises are either RHS grammar symbols together with variables for their attributes ($\mathsf{Element}$) or semantic rules ($\mathsf{Computation}$). The variables are subdivided into inputs and outputs (corresponding to inherited and synthesized positions as far as $\mathsf{Element}$ is concerned). The notation $s(v_1, \ldots, v_m) \to (v'_1, \ldots, v'_n)$ is used to construct parameterized symbols like in extended attribute grammars [WM77] or grammars of syntactical functions [Rie79]. The arrow $\to$ is used to separate inputs and outputs.[3]
- Semantic copy rules are represented by unifying variables of a rule accordingly. "Proper"

[3]Note that the arrow and the brackets enclosing the outputs are omitted if there are no outputs. The brackets enclosing the inputs are omitted if there are no inputs. Note also that the sorts of the parameters in our examples are assumed to be defined by the stems of the variable identifiers.

semantic rules $a_0 := f(a_1, \ldots, a_n)$ are represented as a computation with the variable corresponding to a_0 as output and the variables corresponding to $a_1, \ldots, a_n$ as inputs.

We should justify some details of the assumed representation:

- In the common AG notation, grammar symbols are associated with attributes, whereas we consider parameterized (grammar) symbols. The parameter positions correspond to the attributes, whereas the sorts of the parameters correspond to the attribute types. Since sorts are used to refer to parameter positions, we can also say that the sorts correspond to the attributes. This representation is useful in meta-programs because there is no need to traverse the semantic rules or to maintain separate attribute declarations in order to obtain the attributes associated with a grammar symbol.
- In the declarative reading the order in which computations are written down is totally irrelevant. In contrast, we consider *sequences* of premises in the definition of the domain Rule without separating parameterized grammar symbols and computations from each other. It will be illustrated in Section 4 that such a representation, where computations can be inserted before or after a certain premise, is useful to deal with data-flow issues in meta-programs.[4]
- An AG is represented as a sequence of rules according to the definition of Rules. One could argue that it would be more abstract to consider rather sets of rules. By insisting on sequences we can maintain an order of rules during the transformation of some given rules. Thereby, readability is improved. Moreover, the operational semantics of some AG specification language is possibly sensible to the order of rules.

Example 1 Rule [assign] from Figure 2 is represented as target program fragment as follows:

$$[\mathsf{assign}] \quad \mathit{statement} \to (\mathsf{VUSES}) \; : \; \begin{array}{l} \mathit{vuse} \to (\mathsf{ID}), \mathit{expression} \to (\mathsf{VUSES}_2), \\ \{\mathsf{ID}\} \to (\mathsf{VUSES}_1), \mathsf{VUSES}_1 \cup \mathsf{VUSES}_2 \to (\mathsf{VUSES}). \end{array}$$

In AG jargon VUSES_1 can be regarded as a local attribute. $\{\_\}$ is a unary semantic function symbol, whereas $\_ \cup \_$ is a binary semantic function symbol. $\diamondsuit$

2.2. Proper target program fragments

The above domains are restricted to obtain domains of *proper* fragments. A proper fragment is expected to satisfy certain properties such as that it is well-typed and that it is constructed from proper fragments. For “complete” programs additional completeness properties can be relevant, e.g. completeness of the data-flow in the sense of a well-formed and non-circular AG. The framework is established in a way that meta-programs can only observe and generate proper fragments rather than arbitrary representations. Technically, inference rules are given to obtain the domains $\mathsf{Rules} \subset \overline{\mathsf{Rules}}$, $\mathsf{Rule} \subset \overline{\mathsf{Rule}}, \ldots$; refer to Figure 5[5]. Consider for example the inference rule [Rules] in Figure 5. Its premises state the following properties for proper sequences of rules:

- The single rules must be proper rules themselves.
- The tags must be pairwise distinct.
- The types of the rules must be compatible.

[4] For notational convenience: Since only L-attributed AGs are used in this paper it is assumed to place computations in a way that the left-to-right evaluation is reflected.

[5] π_i denotes the i-th projection for products and sequences. $\#s$ denotes the length of the sequence s.

$$\frac{\begin{array}{ll} & \overline{r}_i \in \mathsf{Rule} \text{ for } i = 1,\ldots,n \\ \wedge & \pi_{\mathsf{Tag}}(\overline{r}_i) \neq \pi_{\mathsf{Tag}}(\overline{r}_j) \text{ for } i,j = 1,\ldots,n, i \neq j \\ \wedge & \exists\, \Sigma : (\mathcal{WT}_{\mathsf{Rule}}(\Sigma, \overline{r}_i) \text{ for } i,\ldots,n) \end{array}}{\langle \overline{r}_1,\ldots,\overline{r}_n\rangle \in \mathsf{Rules}} \quad [\mathsf{Rules}]$$

$$\frac{\begin{array}{ll} & \overline{r} \in \overline{\mathsf{Rule}} \text{ is of the form } \langle t, \overline{e}_0, \langle \overline{e}_1,\ldots,\overline{e}_n\rangle\rangle, n \geq 0 \\ \wedge & \overline{e}_0 \in \mathsf{Element} \\ \wedge & (\overline{e}_i \in \mathsf{Element} \vee \overline{e}_i \in \mathsf{Computation}) \text{ for } i = 1,\ldots,n \\ \wedge & \exists\, \Sigma : \mathcal{WT}_{\mathsf{Rule}}(\Sigma, \overline{r}) \end{array}}{\overline{r} \in \mathsf{Rule}} \quad [\mathsf{Rule}]$$

$$\frac{\begin{array}{ll} & \overline{e} \in \overline{\mathsf{Element}} \text{ is of the form } \langle n, in, out\rangle \\ \wedge & \pi_i(in) \in \mathsf{Parameter} \text{ for } i = 1,\ldots,\#in \\ \wedge & \pi_i(out) \in \mathsf{Parameter} \text{ for } i = 1,\ldots,\#out \\ \wedge & \exists\, \Gamma, \Sigma : \mathcal{WT}_{\mathsf{Element}}(\Gamma, \Sigma, \overline{e}) \end{array}}{\overline{e} \in \mathsf{Element}} \quad [\mathsf{Element}]$$

$$\frac{\ldots}{\overline{e} \in \mathsf{Computation}} \quad [\mathsf{Computation}]$$

$$\frac{\begin{array}{ll} & \overline{p} \in \overline{\mathsf{Parameter}} \\ \wedge & \exists\, \Gamma, \Sigma : \mathcal{TYPE}_{\mathsf{Parameter}}(\Gamma, \Sigma, \overline{p}) = \pi_{\mathsf{Sort}}(\overline{p}) \end{array}}{\overline{p} \in \mathsf{Parameter}} \quad [\mathsf{Parameter}]$$

Figure 5: Properties of target program fragments

Let us consider well-typedness slightly more in detail. We assume the following domains[6]:

$$\begin{array}{lcccl} \mathsf{Sigma} & \subseteq & \overline{\mathsf{Sigma}} & = & \mathcal{P}(\overline{\mathsf{Profile}}) \\ \mathsf{Profile} & \subseteq & \overline{\mathsf{Profile}} & = & \mathsf{Symbol} \times \mathsf{Sort}^\star \times \mathsf{Sort}^\star \\ \mathsf{Symbol} & = & & & \mathsf{Name} + \mathsf{Operator} + \cdots \end{array}$$

Thus, symbols get associated profiles in the sense of directional many-sorted types, i.e. there are some input and some output positions each of a certain sort. In certain instances, proper profiles have to be restricted. Signatures Σ are (finite) subsets of Profile. In the AG terminology the signature of some rules corresponds to the association of grammars symbols with attributes and to the profiles of the semantic function symbols. Again restrictions might be appropriate in certain instances. If overloading, for example, should be prohibited, a proper signature Σ must satisfy that $\forall p, p' \in \Sigma : \pi_{\mathsf{Symbol}}(p) = \pi_{\mathsf{Symbol}}(p') \Rightarrow p = p'$. The type system for all the fragment types is defined in terms of straightforward type rules which are omitted here for brevity. $\mathcal{TYPE}_{\mathsf{Rules}}(rs)$ is used in the paper to denote the type (i.e. the signature) of some rules rs. The type system can be refined to cope with constructs and properties which are specific to particular instances.

Finally, completeness of target programs should be discussed; refer also to Figure 6. The inference rule $[\mathcal{COMPLETE}]$ states only one basic completeness criterion concerning the data flow. Notions which can be used to refine $[\mathcal{COMPLETE}]$ and $[\mathcal{DFC}]$ are listed in Figure 6 as well. We could be interested, for example, in reducedness in the context-free sense, where the relation $\mathcal{COMPLETE}$ would have to be invoked with further parameters such as the axiom of the grammar. The minimal requirement for a complete data flow ($\mathcal{DFC}$—data-flow completeness) is that for each rule r the *applied* variable occurrences ($\mathcal{AO}(r)$) are contained in the *defining* variable occurrences ($\mathcal{DO}(r)$). Applied variable occurrences are variables on applied positions, that is to say output positions of the conclusion and input positions of the premises; dually for defining positions. The idea behind these terms is that the variables with occurrences on applied positions are expected to be "computed" in terms of variables with occurrences on the defining positions. These terms are used in much the same

[6] $\mathcal{P}$ denotes the power set constructor. We are mainly concerned with finite subsets in the paper.

$$\frac{\mathcal{DFC}(\overline{rs})}{\mathcal{COMPLETE}(\overline{rs},\ldots)} \quad [\mathcal{COMPLETE}]$$

$$\frac{\mathcal{AO}(\overline{r}_i) \subseteq \mathcal{DO}(\overline{r}_i) \text{ for } i = 1,\ldots,n}{\mathcal{DFC}(\langle \overline{r}_1,\ldots,\overline{r}_n\rangle)} \quad [\mathcal{DFC}]$$

Some possible refinements

- non-circularity in AGs
- call-correctness in logic programs
- unknowns in natural semantics
- reducedness in the context-free sense

Figure 6: Completeness of programs

way in extended attribute grammars [WM77]. Similar terms are introduced in [AC90] in the context of natural semantics. Thereby, we may speak of *undefined* and *unused* variables, where a variable v is undefined in the rule r if $v \in \mathcal{AO}(r) \setminus \mathcal{DO}(r)$; dually for unused variables.

2.3. Functional meta-programs

To obtain a meta-programming language, it is proposed to embed the data types for meta-programming into a typed λ-calculus. Functional meta-programs are preferred because of the applicability of equational reasoning for proving properties and the suitability of higher-order functional programming to write abstract program manipulations. Besides the embedded data types, the following specification language constructs are assumed in the resulting functional meta-program calculus:

- *foldl* / *foldr*, non-recursive / recursive *let*,
- the Boolean data type and the conditional $b \to e_1, e_2$,
- products ($\times$), sequences (*), sets ($\mathcal{P}$) and maybe-types ($D? = D + \{?\}$),
- an error element $\top$ for strict error propagation,
- impure constructs to generate fresh variables and symbols.

The error element $\top$ is regarded as an element of any type. Embedding the data types for meta-programming the application of a basic operation, e.g. for the construction of a fragment, returns $\top$ whenever the underlying operation is not defined. Evaluating a term is strict w.r.t. $\top$ with the common exception of the conditional. It is assumed in the sequel that $\mathcal{DEF}(e)$ means that the evaluation of e terminates and returns a proper value (i.e. a value $\neq \top$).

2.4. Properties of meta-programs

Certain properties of meta-programs which are useful to characterize operators and to facilitate well-founded program manipulation are considered. In the sequel the term *transformation* refers to functions on Rules and the type definition $\mathsf{Trafo} = \mathsf{Rules} \to \mathsf{Rules}$ is assumed.

The first definition concerns (α[7]-) total transformations. In general, a transformation does not need to be total because of partial fragment constructors and $\top$. However, for many operators, we can show that they are total.

Definition 1 $f \in \mathsf{Trafo}$ is *α-total* if $\forall rs \in \alpha \subseteq \mathsf{Rules}$: $\mathcal{DEF}(f(rs))$. $\Diamond$

[7] If some property holds only for some inputs α, the property is qualified with α.

Let us consider now a simple preservation property, that is to say type preservation. It is often desirable to keep the output of a transformation compatible (i.e. interchangeable as far as the profiles of the symbols are concerned) with the input. In the AG terminology type preservation means that the association of grammar symbols and attributes and the profiles of the semantic function symbols are exactly preserved.

Definition 2 $f \in \mathsf{Trafo}$ is α-*type-preserving* if $\forall rs \in \alpha \subseteq \mathsf{Rules}$:

$$\mathcal{DEF}(f(rs)) \Rightarrow \mathcal{DEF}(\mathcal{TYPE}_{\mathsf{Rules}}(rs) \sqcup \mathcal{TYPE}_{\mathsf{Rules}}(f(rs))).$$

$\diamondsuit$

Another simple preservation property is skeleton preservation where the notion skeleton corresponds to the notion of the underlying context-free grammar as far as AGs are concerned. Technically, the skeleton of some rules rs denoted by $\mathcal{SKELETON}(rs)$ is obtained by abstracting away all parameters and computations. We omit a formal definition. Obviously, it is a desirable property for transformations focusing on attributes and semantic rules that they do not modify the underlying CFG. Moreover, the property facilitates composition based on the superposition of rules with the same shape. Furthermore, skeleton preservation is necessary to be able to abstract from the skeleton in certain situations.

Definition 3 $f \in \mathsf{Trafo}$ is α-skeleton-preserving if $\forall rs \in \alpha \subseteq \mathsf{Rules}$:

$$\mathcal{DEF}(f(rs)) \Rightarrow \mathcal{SKELETON}(rs) = \mathcal{SKELETON}(f(rs)).$$

$\diamondsuit$

Let us consider a more advanced preservation property. If a given target program is adapted, for example, to cope with some additional computational aspects, the original computational behaviour mostly must be preserved. There are several transformations which satisfy a certain "syntactical" property, that is to say the input of the transformation can be regarded as a projection of the output, where projection means that premises and parameter positions might be removed and some occurrences of variables might be replaced by fresh variables. Such transformations preserve computational behaviour because in some sense the given behaviour is extended and possibly further constrained but not adapted in any more specific sense.

Definition 4 Given $rs, rs' \in \mathsf{Rules}$, rs is a *projection* of rs' if

1. $\forall \tau \in \mathcal{TYPE}_{\mathsf{Rules}}(rs) : \exists \tau' \in \mathcal{TYPE}_{\mathsf{Rules}}(rs') : \tau$ is a projection of τ', i.e.
   if $\tau' = s\ \sigma_1^{\downarrow} \times \cdots \times \sigma_n^{\downarrow} \to \sigma_1^{\uparrow} \times \cdots \sigma_m^{\uparrow}$ then $\exists in_1, \ldots, in_q, out_1, \ldots, out_r$ such that
   - the in_i are pairwise distinct and the out_j are pairwise distinct,
   - each $in_i \in \{1, \ldots, n\}$ and each $out_j \in \{1, \ldots, m\}$ and
   - $\tau = s\ \sigma_{in_1}^{\downarrow} \times \cdots \times \sigma_{in_q}^{\downarrow} \to \sigma_{out_1}^{\uparrow} \times \cdots \times \sigma_{out_r}^{\uparrow}$

   for $i = 1, \ldots, n$, $j = 1, \ldots, m$.
2. $\forall r \in rs$: $\exists r' \in rs'$: $\pi_{\mathsf{Tag}}(r) = \pi_{\mathsf{Tag}}(r')$ and there is a type-consistent substitution θ such that $\theta(\pi_{\mathsf{Conclusion}}(r)) = \Pi(\pi_{\mathsf{Conclusion}}(r'))$ and $\theta(p_1) = \Pi(p'_{w_1})$, ..., $\theta(p_u) = \Pi(p'_{w_u})$, where
   - p_1, ..., p_u are the premises of r, whereas p'_1, ..., p'_n are the premises of r',
   - w_1, ..., w_u are some natural numbers with $1 \le w_1 < \cdots < w_u \le n$ and
   - Π is the function on parameterized symbols projecting parameters according to (1.).

$f \in \mathsf{Trafo}$ is α-*projection-preserving* if $\forall rs \in \alpha \subseteq \mathsf{Rules}$:

$$\mathcal{DEF}(f(rs)) \Rightarrow rs \text{ is a projection of } f(rs).$$

$\diamondsuit$

Finally, two properties concerning $\mathcal{DFC}$ (the minimum requirement for the completeness of the data-flow; refer to Figure 6) are in place. Consider a transformation which preserves $\mathcal{DFC}$ in the sense that $\forall rs \in \mathsf{Rules} : \mathcal{DFC}(rs) \Rightarrow \mathcal{DFC}(f(rs))$ provided the result is defined. Such a preservation property is too weak to characterize transformations w.r.t. $\mathcal{DFC}$ because it does not apply to situations where $\mathcal{DFC}(rs)$ is not satisfied in intermediate results within a compound transformation. The following definition is useful to characterize transformations w.r.t. $\mathcal{DFC}$ in a more general sense.

Definition 5 Let $\mathcal{F}$ be a family $\{f_i \in \mathsf{Trafo}\}_{i \in \mathcal{I}}$ of transformations.

- $f \in \mathsf{Trafo}$ is *α-$\mathcal{DFC}$-preserving w.r.t. $\mathcal{F}$* if $\forall rs \in \alpha \subseteq \mathsf{Rules}$: $\forall i_1, \ldots, i_n \in \mathcal{I}$:
$$(\mathcal{DEF}(f^\star(rs)) \wedge \mathcal{DEF}(f^\star(f(rs)))) \Rightarrow (\mathcal{DFC}(f^\star(rs)) \Rightarrow \mathcal{DFC}(f^\star(f(rs)))),$$
where $f^\star$ denotes $f_{i_n} \circ \cdots \circ f_{i_1}$.
- The *α-$\mathcal{DFC}$-preservation w.r.t. $\mathcal{F}$ for $f \in \mathsf{Trafo}$ is recovered by $f' \in \mathsf{Trafo}$* if $\forall rs \in \alpha$: $\forall i_1, \ldots, i_n \in \mathcal{I}$: $\forall k \in \{1, \ldots, n\}$:
$$(\mathcal{DEF}(f^\star(rs)) \wedge \mathcal{DEF}(f'^\star(f(rs)))) \Rightarrow (\mathcal{DFC}(f^\star(rs)) \Rightarrow \mathcal{DFC}(f'^\star(f(rs)))),$$
where $f^\star$ denotes $f_{i_n} \circ \cdots \circ f_{i_1}$, whereas $f'^\star$ denotes $f_{i_n} \circ \cdots \circ f_{i_{k+1}} \circ f' \circ f_{i_k} \cdots \circ f_{i_1}$.

$\diamondsuit$

Note that the above weak characterization is captured by $\mathcal{DFC}$-preservation w.r.t. $\emptyset$. Recoverability of α-$\mathcal{DFC}$-preservation for f by f' means that f and f' can be composed in a sense to construct an α-$\mathcal{DFC}$-preserving transformation. This property is useful for non-$\mathcal{DFC}$-preserving transformations because it tells that f' compensates for the undefined variables introduced by f. In the paper we assume that $\mathcal{F}$ in Definition 5 corresponds to the set of transformations which are derivable as instances of the operators presented in the paper.

3. Unbundling roles of program synthesis

A few operators for program synthesis are introduced below. The actual selection is example-driven, i.e. the described operators suffice to derive the AG in Figure 2 from the CFG in Figure 1. The emphasis is on the properties of the operators facilitating well-founded transformation. The corresponding operators are meant to be orthogonal in the sense that they model *basic* roles which are sufficient to derive concepts at a higher-level of abstraction, e.g. remote access. A more exhaustive operator suite is developed in [Läm99b] including the actual definition of the operators by means of meta-programs.

3.1. Adding parameter positions

The operator **Add** _ : $\mathsf{Position} \to \mathsf{Trafo}$ with $\mathsf{Position} = \mathsf{Io} \times \mathsf{Symbol} \times \mathsf{Sort}$, $\mathsf{Io} = \{\mathbf{Input}, \mathbf{Output}\}$ is used to add parameter positions to symbols. Consider the transformation **Add** $\langle \mathbf{Input}, s, \sigma \rangle$ applied to some element $s'(p_1, \ldots, p_n) \to (p'_1, \ldots, p'_m)$. The element keeps unchanged if $s \neq s'$. Otherwise it is transformed to $s(p_1, \ldots, p_n, v) \to (p'_1, \ldots, p'_m)$, where v is a fresh variable of sort σ; similarly for adding output positions. All conclusions and premises are transformed in that way.

Example 2 We want to approach to a synthesis of the AG in Figure 2 starting from the CFG in Figure 1. The first trivial step is to add the corresponding terminal attributes for the symbols *vdec* and *vuse* (variable identifiers in the declaration part resp. statement part) by means of the transformation **Add** $\langle$**Output**$, vuse, ID\rangle \circ$ **Add** $\langle$**Output**$, vdec, ID\rangle$. ◊

Proposition 1 **Add** $\langle io, s, \sigma\rangle$ with $io \in \mathsf{Io}$, $s \in \mathsf{Symbol}$, $\sigma \in \mathsf{Sort}$ is total, projection-preserving, skeleton-preserving, but it is neither type-preserving nor $\mathcal{DFC}$-preserving. ◊

The operator **Add** is overloaded to add several positions at once, i.e.:

$$\textbf{Add } \langle\langle io_1, s_1, \sigma_1\rangle, \ldots, \langle io_n, s_n, \sigma_n\rangle\rangle = \textbf{Add } \langle io_n, s_n, \sigma_n\rangle \circ \cdots \circ \textbf{Add } \langle io_1, s_1, \sigma_1\rangle$$

Moreover, an auxiliary operator **Positions _ For _ Of Sort _** : $\mathsf{Io} \times \mathcal{P}(\mathsf{Symbol}) \times \mathsf{Sort} \to \mathsf{Position}^\star$ for the construction of positions all with the same Io and Sort component is needed, i.e.:

$$\textbf{Positions } io \textbf{ For } \{s_1, \ldots, s_n\} \textbf{ Of Sort } \sigma = \langle\langle io, s_1, \sigma\rangle, \ldots, \langle io, s_n, \sigma\rangle\rangle$$

Example 3 We continue Example 2 by adding the auxiliary positions of sort VUSES, which are used in Figure 2 to accumulate all variable identifiers used in the statement part. The following transformation adds these positions:

Add Positions Output For $\{statement\_part, statements, statement, expression\}$ **Of Sort** VUSES

Due to the added parameter positions, the rule [assign], for example, has the following intermediate form in the notation of target program fragments:

[assign] $\quad statement \to (\mathsf{VUSES}) \quad : \quad vuse \to (\mathsf{ID}), expression \to (\mathsf{VUSES}')$.

The final form of the rule [assign] was shown in Example 1. The positions of sort VDECS can be added in a similar way. ◊

3.2. Inserting constant computations

The operator **Default For _ By _** : $\mathsf{Sort} \times \mathsf{Operator} \to \mathsf{Trafo}$ facilitates the elimination of undefined variables by the insertion of "constant computations", i.e. premises with no inputs and one output. Consider the transformation **Default For** σ **By** o applied to the rule r. Let $v_1, \ldots, v_n$ be all the undefined variables of sort σ in r. The premises $o \to (v_1)$, …, $o \to (v_n)$ are inserted into r.

Example 4 **Default For** VUSES **By** $\emptyset$ is useful to add the computations for the synthesized attributes of sort VUSES in the rules [skip] and [const] in Figure 2. Actually, the attributes are identified with the empty set. ◊

Proposition 2 **Default For** σ **By** o with $\sigma \in \mathsf{Sort}$, $o \in \mathsf{Operator}$ is α-total with $\alpha \subseteq \mathsf{Rules}$ such that $\forall rs \in \alpha$: $\mathcal{DEF}(\mathcal{TYPE}_{\mathsf{Rules}}(rs) \sqcup \{o :\to \sigma\})$, type-, skeleton-, projection- and $\mathcal{DFC}$-preserving. ◊

Proposition 3 The $\mathcal{DFC}$-preservation of **Add** $\langle io, add, \sigma\rangle$ is recovered by **Default For** σ **By** by, where $io \in \mathsf{Io}$, $add \in \mathsf{Symbol}$, $by \in \mathsf{Operator}$, $\sigma \in \mathsf{Sort}$. ◊

3.3. Inserting unary conditions

The operator **Use _ By _** : $\mathsf{Position} \times \mathsf{Operator} \to \mathsf{Trafo}$ facilitates the insertion of unary conditions (i.e. premises with one input and no outputs). Consider the transformation **Use** $\langle io, s, \sigma\rangle$ **By** o applied to the rule r. For each parameter p on a defining position matching with $\langle io, s, \sigma\rangle$ a corresponding premise $o(p)$ is inserted.

Example 5 To continue the synthesis of Figure 2 it is explained how for any occurrence *vuse*.ID the corresponding (singleton) set of variable identifiers is derived. The corresponding computations can be added to the rules [assign] and [var] by the following transformation:

$$\textbf{Add}\ \langle \textbf{Output}, \{\_\}, \mathsf{VUSES}\rangle \circ \textbf{Use}\ \langle \textbf{Output}, \mathit{vuse}, \mathsf{ID}\rangle\ \textbf{By}\ \{\_\}$$

Thus, to insert the corresponding unary computations, first a unary condition is added (**Use**), second an output position is added (**Add**). As far as the rule [assign] is concerned, for example, the following intermediate form is achieved:

$$[\mathsf{assign}] \quad \mathit{statement} \to (\mathsf{VUSES}) \ : \ \mathit{vuse} \to (\mathsf{ID}), \mathit{expression} \to (\mathsf{VUSES}_1), \{\mathsf{ID}\} \to (\mathsf{VUSES}_2).$$

Note that the above rule is not yet in the final form (refer to Example 1) because VUSES_1 and VUSES_2 must still be combined to compute VUSES. ♢

Proposition 4 **Use** $\langle io, s, \sigma\rangle$ **By** o with $io \in \mathsf{Io}$, $s \in \mathsf{Symbol}$, $\sigma \in \mathsf{Sort}$, $o \in \mathsf{Operator}$ is α-total with $\alpha \subseteq \mathsf{Rules}$ such that $\forall rs \in \alpha$: $\mathcal{DEF}(\mathcal{TYPE}_{\mathsf{Rules}}(rs) \sqcup \{o : \sigma\})$, type-, skeleton-, projection- and $\mathcal{DFC}$-preserving. ♢

3.4. Pairing unused occurrences

The operator **Reduce _ By _** : $\mathsf{Sort} \times \mathsf{Operator} \to \mathsf{Trafo}$ is used to pair unused variables of a certain sort σ in a dyadic computation deriving a new defining position of sort σ. The purpose of these computations is to reduce any number > 1 of unused variables of sort σ to 1. Consider the transformation **Reduce** σ **By** o applied to the rule r. Let $v_1, \ldots, v_n$ be all the unused variables of sort σ in r. The computations $o(v_1, v_2) \to (v_{n+1})$, $o(v_{n+1}, v_3) \to (v_{n+2})$, …, $o(v_{n+n-2}, v_n) \to (v_{n+n-1})$, are inserted into r, where the variables $v_{n+1}, \ldots, v_{n+n-1}$ are fresh variables of sort σ. Thus, v_{n+n-1} will be the only unused variable of sort σ in the result of the transformation.

Example 6 Example 5 is continued. Several defining occurrences of sort VUSES in a given rule can be combined by the transformation **Reduce** VUSES **By** $\_ \cup \_$. As far as the rule [assign] is concerned, for example, the following intermediate form is achieved:

$$\begin{array}{llll}[\mathsf{assign}] & \mathit{statement} \to (\mathsf{VUSES}) & : & \mathit{vuse} \to (\mathsf{ID}), \mathit{expression} \to (\mathsf{VUSES}_1), \\ & & & \{\mathsf{ID}\} \to (\mathsf{VUSES}_2), \mathsf{VUSES}_1 \cup \mathsf{VUSES}_2 \to (\mathsf{VUSES}').\end{array}$$

Note that the above rule is still not yet in the final form (refer to Example 1) because the variables VUSES′ and VUSES must be unified. ♢

Proposition 5 **Reduce** σ **By** o with $\sigma \in \mathsf{Sort}$, $o \in \mathsf{Operator}$ is α-total with $\alpha \subseteq \mathsf{Rules}$ such that $\forall rs \in \alpha$: $\mathcal{DEF}(\mathcal{TYPE}_{\mathsf{Rules}}(rs) \sqcup \{o : \sigma \times \sigma \to \sigma\})$, type-, skeleton-, projection- and $\mathcal{DFC}$-preserving.

♢

3.5. Inserting copy rules

The operator **From The Left _** : $\mathsf{Sort} \to \mathsf{Trafo}$ facilitates propagation by *copying systematically defining occurrences of a certain sort to undefined variables from left to right.* In AG jargon we would say that copy rules are established. Note that an application of the operator corresponds to the insertion of a potentially unknown number of copy rules. The schema is sufficient to establish certain patterns of propagation, e.g. a bucket brigade, provided the necessary positions have been added in advance. Consider the transformation **From The Left** σ applied to the rule r. Any undefined variable v of sort σ in r is replaced by the first defining variable occurrence v' of sort σ to the left of v. Note that the transformation depends on the actual positions of computations.

Example 7 Example 6 is continued. To transmit the combined sets of variable identifier upward in the decorated syntax tree the transformation **From The Left** VUSES is useful. As far as the rule [assign] is concerned, for example, the above transformation exactly corresponds to the missing step to arrive at the final form shown in Example 1. ♢

Proposition 6 **From The Left** σ with $\sigma \in \mathsf{Sort}$ is total, type-, skeleton-, projection- and $\mathcal{DFC}$-preserving. ♢

4. Investigating combinations of roles

It will be shown that certain extensions of the basic AG paradigm can be modelled by meta-programs at a high level of abstraction. The first subsection provides some auxiliary notions. Then, the reconstruction of concepts facilitating the schematic addition of computations (e.g. propagation patterns, rule models, remote access) is investigated in some detail. Finally, an entirely new notion of schematic adaptation of computations is derived.

4.1. Auxiliary operators

The auxiliary operator **From _ To _ In _** : $\mathcal{P}(\mathsf{Symbol}) \times \mathcal{P}(\mathsf{Symbol}) \times \mathsf{Skeleton} \rightarrow \mathcal{P}(\mathsf{Symbol})$ serves for the computation of closures of symbols in the sense of the reachability relation for context-free grammars. Taking such closures is an important tool because thereby program manipulations may abstract from the underlying skeleton of a target program. Actually, this kind of reachability can be seen as the basis of a simulation of remote access constructs as in *Lido* [KW94] in the basic paradigm.

Obviously, a skeleton $sk \in \mathsf{Skeleton}$ can be regarded as a CFG. Thus, it makes sense to consider the transitive closure $\Rightarrow^+_{sk}$ of the context-free direct derivation relation w.r.t. the grammar sk. **From** *from* **To** *to* **In** *sk* is assumed to compute the set of all symbols $s \in \mathsf{Name}$ satisfying the property $\exists f \in \mathit{from}, \exists t \in \mathit{to} : f \Rightarrow^+_{sk} s \Rightarrow^+_{sk} t$.

Example 8 Example 3 can be improved by abstracting from the underlying CFG, i.e. the closure of symbols contributing to the synthesis is derived from the skeleton:

Add (**Positions Output For** (**From** $\{\mathit{program}\}$ **To** $\{\mathit{vuse}\}$ **In** Figure 1) **Of Sort** VUSES)

♢

The notion of object-oriented CFGs together with attribute inheritance (and maybe rule models) [Kos91] and the notion of abstract symbol computations (together with inheritance) in *Lido* [KW94] are also somehow concerned with the abstraction from the underlying CFG. However, these concepts are more directly concerned with "collective" computations, i.e. computations which are applicable to a set of symbols (in several occurrences maybe) rather than a single symbol. These notions can be easily modelled by using transformations parameterized by symbols and by computing closures of symbols based on a class system defined in one or another way.

Let us consider another technical issue. The scope of transformations frequently needs to be restricted in a certain controlled way. The following forms are needed in the paper (especially in Figure 7). The transformation **Selecting** *ts* **Do** *f* applies f only to those input rules with tags in ts, whereas the other rules are taken over. The transformation **Hiding** s **Do** f renames s to a fresh symbol in the input rules before f is applied. After the transformation, the renaming is nullified. Thereby, a confusion of occurrences of s in the input rules and occurrences inserted by some part of f is avoided. Such a confusion may otherwise cause type conflicts or unintended modifications of previous occurences. The combinators are defined as follows:

$$\begin{array}{rcll} (\textbf{Selecting}\ ts\ \textbf{Do}\ f)\ (rs) & = & (f(rs|_{ts})) \bowtie_{rs} rs|_{\mathcal{TAGS}(rs)\setminus ts} & \\ (\textbf{Hiding}\ s\ \textbf{Do}\ f)\ (rs) & = & [s'/s](f([s/s']rs)) & s' \text{ is a fresh symbol} \end{array}$$

The following notation is assumed:

- $\mathcal{TAGS}(rs)$ denotes the *tags* of rs (i.e. $\{\pi_1(\pi_1(rs)), \ldots, \pi_1(\pi_{\#rs}(rs))\}$).
- $rs|_{ts}$ denotes the sequence of rules from rs with tags in ts.
- $rs|_{ss}$ denotes the sequence of rules from rs with LHS symbols in ss.
- $rs_1 \bowtie rs_2$ denotes the concatenation of the rules rs_1 and rs_2.
- $rs_1 \bowtie_{rs} rs_2$ denotes a permutation of $rs_1 \bowtie rs_2$ such that the order of tags in rs is preserved.
- $[s/s']rs$ denotes the rules obtained from rs by replacing each occurrence of s by s'.

Proposition 7 **Selecting** ts **Do** f and **Hiding** s **Do** f with $ts \in \mathcal{P}(\mathsf{Tag})$, $s \in \mathsf{Symbol}$, $f \in \mathsf{Trafo}$ are skeleton-, type- or projection-preserving resp. if f has the corresponding property.[8] $\diamondsuit$

4.2. Adding computations schematically

Several extensions of the AG paradigm are concerned with the schematic definition of computations, e.g. attribute classes [Le 89, Le 93] in *Olga* (*FNC-2*) [JP91], remote access and symbol computations [KW94] in *Lido* (*Eli*), rule models [KLMM93] in Mj$\not{o}$lner/Orm and templates in *Lisa*'s sense [MZ98] or in the sense of *MAGs* [DC90]. It should be obvious that standard examples like the value distribution pattern, the propagate pattern, the bucket brigade (left) can be easily represented in our framework based on the reachability relation (**From** _ **To** _ **In** _) and the operators **Add** and **From The Left**. That should also explain why the *Including*-construct and chains as in *Lido* can be simulated in our framework; refer to [Läm99b] for details. To simulate rule models the operator **Default** additionally has to be taken into consideration.[9] We want to give a detailed reconstruction of a more intricate construct, that is to say the *Constituents*-construct of *Lido* which cannot be simulated, for example, with rule models at all and in no satisfying manner with MAGs.

Figure 7 defines an operator **Constituents** _._ **With** (_, _, _, _) **For** _ **In** _ modelling the corresponding construct in *Lido*. The first six parameters coincide with the parameters of the construct in *Lido*, whereas the remaining two parameters refer to the symbol of the nodes rooting the accumulation and to the rule, where the remote access takes place. The actual definition should be clear from the examples from the previous section performing all the functionality step by step.

Example 9 The pure AG in Figure 2 is derived from the CFG in Figure 1 using the operator **Constituents**. Indeed, evaluating the following expression returns almost Figure 2:

$$\begin{array}{ll} (& \textbf{Constituents}\ \mathit{vdec}.\mathsf{ID}\ \textbf{With}\ (\mathsf{VDECS}, \_\cup\_, \{\_\}, \emptyset)\ \textbf{For}\ \mathit{declaration\_part}\ \textbf{In}\ [\mathsf{axiom}] \\ & \circ\ \textbf{Constituents}\ \mathit{vuse}.\mathsf{ID}\ \textbf{With}\ (\mathsf{VUSES}, \_\cup\_, \{\_\}, \emptyset)\ \textbf{For}\ \mathit{statement\_part}\ \textbf{In}\ [\mathsf{axiom}] \\ & \circ\ \textbf{Add}\ \langle\langle \textbf{Output}, \mathit{vuse}, \mathit{ID}\rangle, \langle \textbf{Output}, \mathit{vdec}, \mathit{ID}\rangle\rangle \\ ) & (\text{Figure 1}) \end{array}$$

It remains to add the following computation to the rule [axiom]:

$$\mathit{program}.\mathsf{USELESS}\uparrow \quad := \quad \mathit{declaration\_part}.\mathsf{VDECS}\uparrow \setminus \mathit{statement\_part}.\mathsf{VUSES}\uparrow$$

$\diamondsuit$

Proposition 8 **Constituents** $\mathit{rsym}.\mathit{rsort}$ **With** $(\mathit{aux}, \mathit{union}, \mathit{unit}, \mathit{zero})$ **For** for **In** in with $\mathit{rsym}, \mathit{for} \in \mathsf{Name}$, $\mathit{union}, \mathit{unit}, \mathit{zero} \in \mathsf{Operator}$, $\mathit{rsort}, \mathit{aux} \in \mathsf{Sort}$, $\mathit{in} \in \mathsf{Tag}$ is α-total with $\alpha \subseteq \mathsf{Rules}$ such that $\forall rs \in \alpha$: $\mathcal{DEF}(\mathcal{TYPE}_{\mathsf{Rules}}(rs) \sqcup \{\mathit{union} : \mathit{aux} \times \mathit{aux} \to \mathit{aux}, \mathit{unit} : \mathit{rsort} \to \mathit{aux}, \mathit{zero} :\to \mathit{aux}\})$, skeleton-, projection- and $\mathcal{DFC}$-preserving. $\diamondsuit$

There are several proposals to extend the AG formalism with object-oriented notions such as inheritance [Kos91, KW94, MZ98]. Inheritance facilitates "collective semantic rules" (refer to Subsection 4.1) and overriding. We should comment on the simulation of overriding—at least—semantic rules. Obviously, every concrete computation c in an AG ag can be represented as a

[8] The precondition for f so that the combined transformation will be total is slightly more involved.

[9] Actually, to simulate the most general kind of rule models (in contrast to the simple examples in [KLMM93]) an elaborate variant of **Default** is needed.

$\lambda rsym$	:	Name.	% the symbol of the remote nodes
$\lambda rsort$	:	Sort.	% the sort of the remote attributes
λaux	:	Sort.	% the auxiliary sort for combined occurrences
$\lambda union$	:	Operator.	% binary combination of (combined) occurrences
$\lambda unit$	:	Operator.	% to qualify single occurrences for combination
$\lambda zero$	:	Operator.	% to represent zero occurrences
λfor	:	Name.	% the symbol of nodes intended to carry the combined occurrences
λin	:	Tag.	% the rule intended to perform remote access
λrs	:	Rules.	% the input rules

	Let $sk = \mathcal{SKELETON}(rs)$ **In**	% the skeleton preserved by the schema
	Let $cl =$ (**From** $\{for\}$ **To** $\{rsym\}$ **In** $sk) \cup \{for\}$ **In**	% symbols contributing to the synthesis
5)	(**Default For** *aux* **By** *zero*	% deal with empty occurrences in a subtree
4)	$\circ$ **From The Left** *aux*	% propagate combined occurrences
3)	$\circ$ **Selecting** $\mathcal{TAGS}(rs) \setminus \{in\}$ **Do**	% restrict scope
	Reduce *aux* **By** *union*	% pairwise combination of occurrences
2)	$\circ$ **Add Positions Output For** *cl* **Of Sort** *aux*	% add positions for combined occurrences
1)	$\circ$ **Selecting** $\mathcal{TAGS}(rs\|_{cl})$ **Do**	% restrict scope
	Hiding *unit* **Do** (	% incremental construction of premises
1*b*)	**Add** $\langle$**Output**, *unit*, *aux*$\rangle$	% transform conditions into computations
1*a*)	$\circ$ **Use** $\langle$**Output**, *rsym*, *rsort*$\rangle$ **By** *unit*	% add conditions for the remote nodes
	)	
	)	
	(rs)	% apply the composed transformation to the input rules

Figure 7: Reconstruction of the *Constituents*-construct

transformation ξ_c intended to insert c into the corresponding rule in a target program with the skeleton of *ag*. A schema s (e.g. a bucket brigade) from a certain class can also be represented as a corresponding transformation ξ_s intended to insert all the derived computations. The simplest form of overriding assumes that concrete computations have a higher priority than computations derivable from a schema. In our functional framework this idea is very easily realized. Consider an AG specification *ag* dealing with the underlying CFG (skeleton) *sk*, some attributes $a_1, \ldots, a_m$, concrete computations $c_1, \ldots, c_n$ and schemata (templates, rule models, symbol computations) $s_1, \ldots, s_k$. The corresponding target program representation is obtained by the following application:

(	$\xi_{s_k} \circ \cdots \circ \xi_{s_1}$	% insert the computations according to $s_1, \ldots, s_k$
$\circ$	$\xi_{c_n} \circ \cdots \circ \xi_{c_1}$	% insert the computations $c_1, \ldots, c_n$
$\circ$	**Add** $\langle \ldots \rangle$	% add the positions according to $a_1, \ldots, a_m$
)	(sk)	

It is assumed that the transformations modelling the schemata do not insert computations for attributes which are already defined by other computations. Indeed, operators like **Default** and **From The Left** are defined in this way. Note also that it is a very pleasant consequence of our analysis that this kind of overriding preserves computational behaviour if the ξ_{s_k}, …, ξ_{s_1} are projection-preserving.

4.3. Adapting computations schematically

All the concepts referenced in the previous subsection do not facilitate the *schematic adaptation* of computations in contrast to overriding. It will be shown that such an adaptation is useful and feasible.

Example 10 Consider the following fragment of an AG describing the interpretation[10] of an assignment and of an expression consisting of a variable. Obviously, the specification does not cope with side-effects during expression evaluation because of the restricted data-flow of the memory (i.e. the nonterminal *expression* does not synthesize a corresponding attribute of sort MEM).

$$
\begin{array}{llcl}
\ldots & & & \\
[\mathsf{assign}] & \mathit{statement} & ::= & \mathit{vuse}\ \mathit{expression} \\
 & \mathit{expression}.\mathsf{MEM}\!\downarrow & := & \mathit{statement}.\mathsf{MEM}\!\downarrow \\
 & \mathit{statement}.\mathsf{MEM}\!\uparrow & := & \mathit{update}(\mathit{statement}.\mathsf{MEM}\!\downarrow, \mathit{vuse}.\mathsf{ID}\!\uparrow, \mathit{expression}.\mathsf{VAL}\!\uparrow) \\
\ldots & & & \\
[\mathsf{var}] & \mathit{expression} & ::= & \mathit{vuse} \\
 & \mathit{expression}.\mathsf{VAL}\!\uparrow & := & \mathit{lookup}(\mathit{expression}.\mathsf{MEM}\!\downarrow, \mathit{vuse}.\mathsf{ID}\!\uparrow) \\
\ldots & & &
\end{array}
$$

Now let us assume that the interpreter definition needs to be adapted in order to cope with side-effects during expression evaluation. Thus, a synthesized attribute of sort MEM must be added to *expression* and the new attribute must be incorporated into the data-flow. As far as the above fragment is concerned, the following variant is needed where the boxes indicate the modified positions:

$$
\begin{array}{llcl}
\ldots & & & \\
[\mathsf{assign}] & \mathit{statement} & ::= & \mathit{vuse}\ \mathit{expression} \\
 & \mathit{expression}.\mathsf{MEM}\!\downarrow & := & \mathit{statement}.\mathsf{MEM}\!\downarrow \\
 & \mathit{statement}.\mathsf{MEM}\!\uparrow & := & \mathit{update}(\boxed{\mathit{expression}.\mathsf{MEM}\!\uparrow}, \mathit{vuse}.\mathsf{ID}\!\uparrow, \mathit{expression}.\mathsf{VAL}\!\uparrow) \\
\ldots & & & \\
[\mathsf{var}] & \mathit{expression} & ::= & \mathit{vuse} \\
 & \mathit{expression}.\mathsf{VAL}\!\uparrow & := & \mathit{lookup}(\mathit{expression}.\mathsf{MEM}\!\downarrow, \mathit{vuse}.\mathsf{ID}\!\uparrow) \\
 & \boxed{\mathit{expression}.\mathsf{MEM}\!\uparrow \quad := \quad \mathit{expression}.\mathsf{MEM}\!\downarrow} & & \\
\ldots & & &
\end{array}
$$

◇

Example 10 illustrates once more that altering design decisions may affect several parts of a specification resulting in a lack of extensibility. Thus, we are looking for concepts modelled by transformations to avoid that specifications need to be rewritten.

The scenario of the above example is paraphrased in more general terms. The corresponding schematic way to adapt computations is concerned with *turning from distribution to accumulation*; refer to Figure 8 for an illustration. To be more precise, the scope of accumulation is extended in the sense that certain symbols additionally synthesize an attribute of the dedicated sort σ. Due to the inserted output positions, there will be unused and undefined occurrences of sort σ. These occurrences should be eliminated by adapting parameter positions of sort σ in such a way that a (depth-first) left-to-right data-flow is re-established.

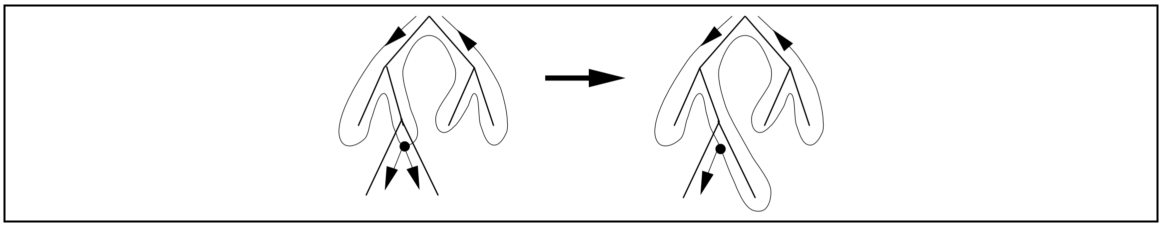

Figure 8: Extending a thread

The following transformation can be applied if certain symbols $ss \in \mathcal{P}(\mathsf{Symbol})$ should participate in accumulation of a data structure of sort σ rather than distribution:

[10]Dynamic semantics is not the standard domain for AGs. However, we can assume a form of AGs more suitable for this domain, e.g. dynamic attribute grammars [PRJD96].

$$\textbf{From The Left } \sigma \circ \textbf{Refresh } \sigma \circ \textbf{Add Positions Output For } ss \textbf{ Of Sort } \sigma$$

Refresh σ denotes a simple transformation refreshing all variables of sort σ, i.e. considering the result of the transformation, any applied occurrence is an undefined occurrence and any defining occurrence is an unused occurrence. Actually, the computations are relocated before in a way that the assigned locations reflect the left-to-right dependencies w.r.t. sort σ.

Proposition 9 **Refresh** σ is total and type-preserving and the $\mathcal{DFC}$-preservation of **Refresh** σ is recovered by **Default For** σ **By** s, where $\sigma \in \mathsf{Sort}$, $s \in \mathsf{Symbol}$. $\diamondsuit$

The transformation **Refresh** σ is obviously not projection-preserving, but the above application stretching the scope of accumulation preserves computational behaviour in another sense if some preconditions are met: The rules to be transformed must show left-to-right dependencies as far as σ is concerned[11] and there must not be any defining occurrences of sort σ in the premises of rules with a LHS symbol in ss. Then, computational behaviour is preserved because for any node the additional synthesized attribute will be equal to the original inherited attribute.

Example 11 The adaptation outlined in Example 10 is performed step by step. We will be only concerned with the rule [assign]. First, the original rule which does not cope with side-effects during expression evaluation is represented as target program fragment:

$$[\mathsf{assign}] \quad \mathit{statement}(\mathsf{MEM}) \to (\mathsf{MEM})' \; : \; \begin{array}{l} \mathit{vuse} \to (\mathsf{ID}), \mathit{expression}(\mathsf{MEM}) \to (\mathsf{VAL}), \\ \mathit{update}(\mathsf{MEM}, \mathsf{ID}, \mathsf{VAL}) \to (\mathsf{MEM}'). \end{array}$$

Inserting an output position of sort MEM for *expression*, the rule takes the following form:

$$[\mathsf{assign}] \quad \mathit{statement}(\mathsf{MEM}) \to (\mathsf{MEM}') \; : \; \begin{array}{l} \mathit{vuse} \to (\mathsf{ID}), \mathit{expression}(\mathsf{MEM}) \to (\mathsf{VAL}, \boxed{\mathsf{MEM}''}) \\ \mathit{update}(\mathsf{MEM}, \mathsf{ID}, \mathsf{VAL}) \to (\mathsf{MEM}'). \end{array}$$

Refreshing all parameters of sort MEM the rule takes the following form:

$$[\mathsf{assign}] \quad \mathit{statement}(\boxed{\mathsf{MEM}_1}) \to (\boxed{\mathsf{MEM}_6}) \; : \; \begin{array}{l} \mathit{vuse} \to (\mathsf{ID}), \mathit{expression}(\boxed{\mathsf{MEM}_2}) \to (\mathsf{VAL}, \boxed{\mathsf{MEM}_3}) \\ \mathit{update}(\boxed{\mathsf{MEM}_4}, \mathsf{ID}, \mathsf{VAL}) \to (\boxed{\mathsf{MEM}_5}). \end{array}$$

Finally, the refreshed parameters of sort *MEM* are identified to encode a data-flow from left to right. Note that the final form was also shown in Example 10 in the common AG notation.

$$[\mathsf{assign}] \quad \mathit{statement}(\boxed{\mathsf{MEM}}) \to (\boxed{\mathsf{MEM}''}) \; : \; \begin{array}{l} \mathit{vuse} \to (\mathsf{ID}), \mathit{expression}(\boxed{\mathsf{MEM}}) \to (\mathsf{VAL}, \boxed{\mathsf{MEM}'}) \\ \mathit{update}(\boxed{\mathsf{MEM}'}, \mathsf{ID}, \mathsf{VAL}) \to (\boxed{\mathsf{MEM}''}). \end{array}$$

$\diamondsuit$

We should explain why schematic adaptation has not been suggested in other frameworks so far. MAGs, rule models and symbol computations can be overridden ("pointwise") by concrete computations but not vice versa. Indeed, certain technical problems arise when a given complete AG specification should be adapted schematically. It would be straightforward to override *all* computations computing relevant attributes by the new computations derived from the schema, but usually some computations should be retained; maybe they have to be adapted, e.g. the computation modifying the memory in rule [assign] in Example 10. That seems to be an intricate problem. We can retain and adapt computations in our approach because of the following two special phases. In the first phase the data-flow is made totally undefined as far as the dedicated sort is concerned (**Refresh**). Copy rules get lost, but not proper computations including their left-to-right order. In the second phase the data-flow is re-established according to the schema. The retained computations contribute to the data-flow. The two phases are illustrated in Example 11 where the intermediate result is shown.

[11] i.e.: $rs = (\textbf{From The Left } \sigma \circ \textbf{Refresh } \sigma)\ (rs)$

Lisa [ZKM97, MZ98] supports a sophisticated concept to extend an AG. One can, for example, override computations of the inherited AG by other computations possibly expressed in terms of templates. Note that such an adaptation can be simulated in our framework if we introduce a rather simple (non-projection preserving) operator to remove computations. Nevertheless, this kind of adaptation still forces one to override all the single relevant computations "by hand".

5. Concluding remarks

First, the results of the paper are concluded. Afterwards, related work is considered. Finally, a few remarks on future work are provided.

5.1. Results

We developed a detailed formal framework for functional meta-programs which is suitable to instantiate transformational programming for AGs. We unbundled and analysed roles present in several extensions of the basic AG paradigm such as remote access [Kas76, Lor77, KW94], symbol computations [KW94], rule models [KLMM93]. The genericity of the framework allows us to take a general point of view, that is to say we do not only reconstruct AG concepts but also concepts proposed for other representatives of the declarative paradigm, e.g. stepwise enhancement [Lak89, KMS96, Nai96].

There are some unique technical contributions. First, we have shown that it is possible to model roles of the synthesis of AGs as skeleton-preserving, projection-preserving and $\mathcal{DFC}$-preserving program transformations. Second, neither the general notion of schematic adaptation, where concrete computations are overriden according to a schema, nor an instance of it have been proposed or implemented before. We give an example where distribution is partially extended to accumulation.

The framework and a superset of the outlined operators have been implemented in $\Lambda\Delta\Lambda$ [HLR97, Läm99b] with applications in language definition based on AGs and operational semantics.

5.2. Related work

Besides the extensions of the basic AG paradigm discussed throughout the paper, there are some further related approaches.

Kiczales et al. recently proposed aspect-oriented programming (AOP) [KLM$^+$97], where special language support is used to *weave* (non-functional) properties—the so-called aspects—into the basic functionality. Standard examples for aspects are error handling and optimization. Implementing aspects in the traditional approach would result in tangled code which is then unclear and hard to modify and adapt because these properties tend to cross-cut the functionality. Meta-programming is a viable option for developing aspect code and performing weaving [FS98, Läm99a].

The *Demeter* Research Group (Karl J. Lieberherr et al.) has developed an extension of object-oriented programming, that is to say adaptive object-oriented programming [Lie95]. The *Demeter* method proposes *class dictionaries* for defining the class structure and *propagation patterns* for implementing the behaviour of the objects. Our approach is similar to that of *Demeter* in that transformations are independent from the actual skeleton and how computational behaviour based on the notion of reachability can be established in concrete target programs.

Sterling's et al. *stepwise enhancement* [Lak89, JS94, KMS96] advocates developing logic programs systematically from skeletons and techniques. Skeletons are (in contrast to our terminology) simple logic programs with a well-understood control flow, whereas techniques are common programming practices. Applying a technique to a program yields a so-called enhancement. Our framework for functional meta-programs and our program manipulations are effective means to develop and to reason about techniques. Kirschbaum, Sterling et al. have shown in [KSJ93] that program maps—a tool similar to our projection-preserving transformations—preserve the computational behaviour of a logic program, if we assume that behaviour is manifested by the SLD computations of the program. Note that our approach is generic and that we make vital use of modes and sorts. Skeletons in the sense of our framework are not instrumented in stepwise enhancement.

5.3. Future work

The paper emphasizes that program transformations can model concepts which are similar to extensions in other specification languages like *Lido* (*Eli*) [KW94], *Olga* (*FNC-2*) [JP91] or *Lisa* [ZKM97, MZ98]. Another point of view emphasizing the aspect of adaptation and to support it by means of a transformational programming environment should receive more attention. There is some related work in this respect, e.g. Attali's et al. environment for program transformation based on the rule-based language *TrfL* for program transformations [APR97] (primary intented for non-declarative programs) or program manipulation systems such as *Translog* [Bru95] and *Spes* [ABFQ92] in the logic programming community.

Another issue concerns the correctness of roles and reconstructions. Although we represent our transformations as functional programs, it is apparently not trivial to provide rigorous proofs for all the propositions we are interested in, e.g. the property of our reconstruction of the *Constituents*-construct to be projection-preserving, or the more general question if it is really correct w.r.t. the primary form in *Lido*. Some propositions should be provable by a rather simple derivation of properties of compound transformations from properties of the underlying transformations. We would like to approach to the more intricate problems by using a theorem prover.

The proposed notion of schematic adaption should be worked out further. The example we have given is obviously tuned towards left-to-right dependencies and it does not illustrate a very broad class of schemata. Other useful scenarios should be identified. Schemata of adaptation such as our example are usually not projection-preserving. Thus, we need to adopt other notions to reason about preservation of computational behaviour.

Finally, we should look for *concrete* suggestions how AG specification formalisms such as *Lido*, *Olga* and *Lisa* could benefit from our analysis and framework.

Acknowledgement

We are very grateful to Isabelle Attali for her constructive review of substantial fragments contributing to the paper. Many thanks also to Marjan Mernik, Didier Parigot and Uwe Kastens for several helpful remarks in the context of the development of the paper.

Bibliography

[ABFQ92] Francis Alexandre, Khadel Bsaies, Jean Pierre Finance, and Alain Quere. Spes: A System for Logic Program Transformation. In A. Voronkov, editor, *Logic Programming and Automated Reasoning, LPAR'92*, volume 624 of *LNCS*, pages 445–447. Springer-Verlag, 1992.

[AC90] Isabelle Attali and Jacques Chazarain. Functional evaluation of strongly non-circular typol specifications. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *LNCS*, pages 157–176. Springer-Verlag, September 1990. Paris.

[AM91] Henk Alblas and Bořivoj Melichar, editors. *Attribute grammars, Applications and Systems, Proceedings of the In ternational Summer School SAGA, Prague, Czechoslovakia*, volume 545 of *LNCS*. Springer-Verlag, June 1991.

[APR97] Isabelle Attali, Valérie Pascual, and Christophe Roudet. A language and an integrated environment for program transformations. Rapport de recherche 3313, INRIA, December 1997.

[Boy96] John Tang Boyland. *Descriptional Composition of Compiler Components*. PhD thesis, University of California, Berkeley, September 1996. Available as technical report UCB//CSD-96-916.

[Boy98] John Tang Boyland. Analyzing Direct Non-local Dependencies. In Kai Koskimies, editor, *Compiler Construction, 7th International Conference, CC'98*, volume 1383 of *LNCS*, pages 31–49. Springer-Verlag, April 1998.

[Bru95] J.J. Brunekreef. Translog, an Interactive Tool for Transformation of Logic Programs. Technical Report P9512, University of Amsterdam, Programming Research Group, December 1995.

[CDPR98] Loic Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Generic programming by program composition (position paper). In *Workshop on Generic Programming*, Marstrand, Sweden, June 1998. conjunction with MPC'98.

[DC90] G.D. Dueck and G.V. Cormack. Modular Attribute Grammars. *The Computer Journal*, 33(2):164–172, 1990.

[DPRJ96] Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Attribute grammars and folds: Generic control operators. Rapport de recherche 2957, INRIA, August 1996.

[DPRJ97] Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Structure-directed genericity in functional programming and attribute grammars. Rapport de Recherche 3105, INRIA, February 1997.

[FMY92] R. Farrow, T.J. Marlowe, and D.M. Yellin. Composable Attribute Grammars. In *Proceedings of 19th ACM Symposium on Principles of Programming Languages (Albuquerque, NM)*, pages 223–234, January 1992.

[FS98] Pascal Fradet and Mario Südholt. AOP: towards a generic framework using program transformation and analysis. In *Position papers of the Aspect-Oriented Programming Workshop at ECOOP'98, http: // wwwtrese . cs . utwente . nl / aop-ecoop98*, July 1998.

[Gie88] R. Giegerich. Composition and Evaluation of Attribute Coupled Grammars. *Acta Informatica 25*, pages 355–423, 1988.

[Hed89] Görel Hedin. An object-oriented notation for attribute grammars. In S. Cook, editor, *Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP'89)*, BCS Workshop Series, pages 329–345. Cambridge University Press, July 1989.

[Hed91] Görel Hedin. Incremental static-semantics analysis for object-oriented languages using door attribute grammars. In Alblas and Melichar [AM91], pages 374–379.

[HLR97] Jörg Harm, Ralf Lämmel, and Günter Riedewald. The Language Development Laboratory ($^{\Lambda}\Delta_{\Lambda}$). In Magne Haveraaen and Olaf Owe, editors, *Selected papers from the 8th Nordic Workshop on Programming Theory, December 4–6, Oslo, Norway, Research Report 248, ISBN 82-7368-163-7*, pages 77–86, May 1997.

[JP91] Martin Jourdan and Didier Parigot. Internals and Externals of the FNC-2 Attribute Grammar System. In Alblas and Melichar [AM91], pages 485–504.

[JS94] Ashish Jain and Leon Sterling. A methodology for program construction by stepwise structural enhancement. Technical Report CES-94-10, Department of Computer Engineering and Science, Case Western Reserve University, June 1994.

[Kas76] Uwe Kastens. Ein Übersetzer-erzeugendes System auf der Basis Attributierter Grammatiken. interner Bericht 10, Fakultät für Informatik, University Karlsruhe, September 1976.

[Kas91] Uwe Kastens. Attribute Grammars in a Compiler Construction Environment. In Alblas and Melichar [AM91], pages 380–400.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *LNCS*, pages 220–242, Jyväskylä, Finland, 9–13 June 1997. Springer-Verlag.

[KLMM93] J. Lindskov Knudsen, M. Löfgren, O. Lehrmann Madsen, and B. Magnusson, editors. *Object-Oriented Environments: The Mjølner Approach.* Prentice Hall, 1993.

[KMS96] M. Kirschenbaum, S. Michaylov, and L.S. Sterling. Skeletons and Techniques as a Normative Approach to Program Development in Logic-Based Languages. In *Proceedings ACSC'96, Australian Computer Science Communications, 18(1)*, pages 516–524, 1996.

[Kos91] Kai Koskimies. Object Orientation in Attribute Grammars. In Alblas and Melichar [AM91], pages 297–329.

[KSJ93] M. Kirschenbaum, L.S. Sterling, and A. Jain. Relating logic programs via program maps. In *Annals of Mathematics and Artifical Intelligence, 8(III-IV)*, pages 229–246, 1993.

[KW94] Uwe Kastens and W.M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica 31*, pages 601–627, 1994.

[Lak89] A. Lakhotia. *A Workbench for Developing Logic Programs by Stepwise Enhancement.* PhD thesis, Case Western Reserve University, 1989.

[Läm99a] Ralf Lämmel. Declarative aspect-oriented programming. In *Proceedings PEPM'99, 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, page 16 pages, January 1999. BRICS Notes Series NS-99-1.

[Läm99b] Ralf Lämmel. *Functional meta-programs towards reusability in the declarative paradigm.* PhD thesis, University of Rostock, Department of Computer Science, 1999.

[Le 89] Carole Le Bellec. Spécification de règles sémantiques manquantes. rapport de DEA, Dépt. d'Informatique, University d'Orléans, September 1989.

[Le 93] Carole Le Bellec. *La généricité et les grammaires attribuées.* PhD thesis, Dépt. d'Informatique, University d'Orléans, 1993.

[Lie95] Karl J. Lieberherr. *Adaptive Object-Oriented Software — The Demeter Method.* PWS Publishing Company, 1995.

[LJPR93] Carole Le Bellec, Martin Jourdan, Didier Parigot, and Gilles Roussel. Specification and Implementation of Grammar Coupling Using Attribute Grammars. In Maurice Bruynooghe and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP '93)*, volume 714 of *LNCS*, pages 123–136, Tallinn, August 1993. Springer-Verlag.

[Lor77] Bernard Lorho. Semantic attributes processing in the system DELTA. In A. Ershov and Cornelius H. A. Koster., editors, *Methods of Algorithmic Language Implementation*, volume 47 of *LNCS*, pages 21–40. Springer-Verlag, 1977.

[MZ98] Marjan Mernik and Viljem Zumer. Incremental language design. *IEE Proc. Softw.*, 145(2–3):85—91, 1998.

[Nai96] Lee Naish. Higher Order Logic Programming in Prolog. In *Proc. Workshop on Multi-Paradigm Logic Programming, JICSLP'96, Bonn*, 1996.

[Paa91] Jukka Paakki. *Paradigms for Attribute-grammar-based Language Implementation.* Ph.D. thesis, Department of Comp. Sc., University of Helsinki, February 1991.

[PRJD96] Didier Parigot, Gilles Roussel, Martin Jourdan, and Etienne Duris. Dynamic Attribute Grammars. In Herbert Kuchen and S. Doaitse Swierstra, editors, *Int. Symp. on Progr. Languages, Implementations, Logics and Programs (PLILP'96)*, volume 1140 of *LNCS*, pages 122–136, Aachen, September 1996. Springer-Verlag.

[Rie79] Günter Riedewald. *Compilerkonstruktion und Grammatiken syntaktischer Funktionen.* Dissertation B, Rechenzentrum der Universität Rostock, 1979.

[SV91] Doaitse Swierstra and Harald Vogt. Higher Order Attribute Grammars. In Alblas and Melichar [AM91], pages 256–296.

[Wat75] David A. Watt. Modular Description of Programming Languages. Technical Report A-81-734, University of California, Berkeley, 1975.

[WM77] D.A. Watt and O.L. Madsen. Extended attribute grammars. Technical Report no. 10, University of Glasgow, July 1977.

[ZKM97] Viljem Zumer, Nikolaj Korbar, and Marjan Mernik. Automatic implementation of programming languages using object-oriented approach. *Journal of Systems Architecture*, 43:203—210, 1997.