Generic Attribute Grammars

João Saraiva[1,2] and Doaitse Swierstra[2]

1: Department of Computer Science, University of Minho,
Campus de Gualtar, 4700 Braga, Portugal
jas@di.uminho.pt
2: Department of Computer Science, University of Utrecht,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
swierstra@cs.uu.nl

Abstract

This paper introduces generic attribute grammars which provide support for genericity, reusability and modularity in attribute grammars. A generic attribute grammar is a *component* that is easily reused, composed and understood. Attribute grammar based systems may be constructed out of a set of generic components, that can be analysed and compiled separately. We show how to generate deforestated attribute evaluators for those components. As a result, redundant intermediate data structures used to *glue* different components are eliminated.

1. Motivation

Recent developments in modern programming languages are providing powerful mechanisms like modularity and polymorphic type systems for abstracting from computations and for structuring programs. Modern applications are nowadays designed and implemented as combinations of several *generic components*, all physically and conceptually separated from each other. The benefits of such an organization are ease of specification, clearness of description, interchangeability between different "plug-compatible" components, reuse of components across applications and separate analysis/compilation of components.

Consider for example the construction of a compiler for a particular language. Such a compiler may be constructed out of a set of generic components, each of which describes a particular subproblem such as name analysis, type checking, register allocation, etc. Furthermore, each component may describe the properties of a subproblem for a large number of languages and not just for the language at hand: a single component may describe the name analysis task for block structured languages. Thus, this "off the shelf" component can be re-used across applications.

Generally, those components are combined by introducing intermediate data structures that act as the *glue*, binding the components of the application together: a component *constructs* (produces) an intermediate data structure which is *used* (consumed) by other components. In a compiler setting the intermediate structure is the abstract syntax tree which glues the compiler task components. Increasing the number of components means simpler and more modular and maintenable code, whereas decreasing the number of components leads to greater runtime efficiency since fewer intermediate data structures have to be used. The long-sought solution to this tension between modularity and reuse on the one hand, and efficiency on the other, is to increase the amount of analysis and program transformation the compiler is performing so that the programmer may write programs as a set of components, confident that the compiler can *fuse* the components together, removing redundant intermediate data structures.

Attribute Grammars (AG) can be split into components in several ways: (a) The components are the non-terminal symbols of the grammar with the associated productions and semantic rules, (b) The components are formed by the different productions associated with specific non-terminal symbols, (c) The components are the semantic domains which are used in the overall computation. For realistic AGs, however, the first two syntactic approaches lead to huge monolithic AGs that are difficult to write and understand since related properties are described in different components, but can only be understood together. A more general and efficient form of modularity is achieved when each semantic domain is encapsulated in a single component [Kas91a, Hen93, FMY92, LJPR93, KW94, SA98]. Traditional AG systems based on such modular descriptions first construct the equivalent monolithic AG and only then produce an implementation for such AG. We call this *syntactic compositionality*. A major disadvantage of this approach is that a single change in one component can render the entire evaluator invalid. Thus the support for *separate analysis and compilation* of components, as provided by modern programming languages, is sought. In this way we aim at *semantic compositionality*.

This paper introduces *Generic Attribute Grammars* (GENAG), which provide a support for genericity, reusability and modularity in the context of attribute grammar based systems. A generic attribute grammar is a *component* which is designed to be easily reused, composed and understood. A generic attribute grammar describes a generic property of an abstract language and has the following properties:

- In a generic attribute grammar some (non-)terminal symbols, from now on called *generic symbols*, may not be defined within the grammar component, and thus are considered as a parameter of the grammar. In other words, generic attribute grammars have "gaps" which are filled later.
- From a generic attribute grammar a *generic attribute evaluator* is derived. In our implementation model a generic evaluator is a purely functional, data type free attribute evaluator as defined in [SS99]. As we will see it is this absence of any explicit data type definitions that makes the evaluators (and the grammars) highly modular and reusable.
- In a generic attribute grammar the productions of a non-terminal symbol may be distributed by different GENAG components. Such components can be analysed and compiled separately, thus giving the sought semantic compositionality.
- A generic attribute grammar can be parameterized with the semantic functions used to compute attribute values.
- Furthermore, deforestated evaluators are used in order to remove redundant intermediate data structures which glue the different components of a GENAG system.

In section 2 we introduce generic attribute grammars and we define a formalism to denote them. We also discuss the circularity of GENAGs and introduce *flow types*. In section 3 generic attribute evaluators are presented. Section 4 discusses the semantic compositionality of GENAGs. Section 5 briefly compares our approach to related work and describes the current implementation. Section 6 contains the conclusions.

2. Generic Attribute Grammars

This section introduces *generic attribute grammars*.

Definition 1 (Generic Context Free Grammar) *A generic context free grammar (*GCFG*) is a triple* $gg = (V, P, S)$. $V = (\Sigma \cup N \cup \mathcal{G})$ *is the* vocabulary, *a finite non-empty set of* grammar symbols.

Σ *is the alphabet,* i.e. *the set of* terminal symbols *and* N *is the non-empty set of* non-terminal symbols. $\mathcal{G}$ *is a finite set of generic symbols.* $P \subset N \times V^*$ *is a finite set of* productions. $S \in (N \cup \mathcal{G})$ *is the* start symbol.

□

A *generic symbol* is a grammar symbol for which there are no productions. As we will explain later, a generic symbol can be the root symbol. We denote a generic symbol $G \in \mathcal{G}$ by $\underline{\mathbf{G}}$. A production $p \in P$ is represented as $X_0 \rightarrow \mathrm{P}\ (X_1\ X_2\ \cdots\ X_s)$. Non-terminal X_0 is called the left-hand side non-terminal or *father* of p and $X_1\ X_2\ \cdots\ X_s$ are the right-hand side symbols or *children* of p.

A generic context free grammar is *complete* if every non-terminal symbol is accessible from the start symbol and can derive a sequence of grammar symbols which contains terminal and generic symbols only. That is, it does not contain any non-terminal symbol.

Definition 2 (Complete Generic Context Free Grammar) *A generic context free grammar* $gg = (V, P, S)$ *is* complete *iff:*

$$\forall_{X \in N}\ \exists_{\mu,\nu,\delta \in (\Sigma \cup \mathcal{G})^*} : S \Rightarrow^* \mu X \nu \ \Rightarrow^* \delta$$

□

A complete GCFG defines a *generic abstract syntax tree.* A generic abstract syntax tree contains *normal leaves* and *generic leaves*, the latter being labelled with a generic symbol.

Definition 3 (Generic Abstract Syntax Tree) *For every production of the form* $X \rightarrow \mathrm{C}\ (Y_1\ \ldots\ Y_n)$ *of a complete generic context free grammar there is a* Generic Abstract Syntax Tree *(*GAST*) labelled* X *with subtrees* $T_1, \ldots, T_n$ *(in that order), where:*

- T_i *is a* normal leaf *labelled* $Y_i \in \Sigma$.
- T_i *is a* generic leaf *labelled* $\underline{Y_i} \in \mathcal{G}$.
- T_i *is a generic abstract syntax tree* $Y_i \in N$.

□

A *Generic Attribute Grammar* is based on a generic context free grammar that is augmented with attributes, attribute equations and semantic functions.

Definition 4 (Generic Attribute Grammar) *A generic attribute grammar is a quadruple* $gag = (gg, A, E, \mathcal{F})$ *where:*

- $gg = (V, P, S)$ *is a generic context free grammar.*
- A *is a finite set of* attributes*, partitioned into sets* $A_{nont}(n)$, $A_{gen}(g)$ *and* $A_{loc}(p)$ *for each* $n \in N$, $g \in \mathcal{G}$ *and* $p \in P$. $A_{nont}(n)$ *and* $A_{gen}(g)$ *are further partitioned into sets* $A_{inh}(n)$, $A_{syn}(n)$ *and* $A_{inh}(g)$, $A_{syn}(g)$*, respectively.*
- $E = \bigcup_{p \in P} E(p)$ *is a finite set of* attribute equations.
- $\mathcal{F} \subseteq E$ *is a finite set of* generic semantic functions.

□

$\mathcal{P} = \mathcal{G} \cup \mathcal{F}$ is the set of *static parameters* of the generic attribute grammar.

We say that a generic attribute grammar is *complete* if its underlying context free grammar is complete and if every local and every *output attribute occurrence* of a production has at least one defining equation and no two equations have the same target. The set of *output attribute occurrences* of a production p contains the synthesized attribute occurrences of the father and the inherited attribute occurrences of the children.

Completeness alone does not guarantee that all the attributes of a generic abstract syntax tree are effectively computable: circular dependencies may occur. Circularities will be discussed in section 2.2. However, if circularities do not occur for any derivable tree, the generic attribute grammar is called *well-defined*. That is, for each generic abstract syntax tree of a GenAg all the attribute instances are effectively computable.

The traditional definition of *well-defined attribute grammars* assumes that the synthesized attributes of terminal symbols are defined by an external module: the lexical analyser [Paa95]. In our definition of well-defined generic attribute grammar we make a similar assumption, that is, the synthesized attributes of the generic symbols are defined by an external module: an *external* generic attribute grammar. Thus, the attribute equations defining the synthesized attributes of a generic symbol are not included in the GenAg which uses such generic symbol.

2.1. Generic Attribute Grammar Specifications

This section describes our notation for generic attribute grammars. We use a *standard* attribute grammar notation: Productions are labelled with a name for future references. Within the attribution rules of a production, different occurrences of the same symbol are denoted by distinct subscripts. Inherited (synthesized) attributes are denoted with the down (up) arrow $\downarrow$ ($\uparrow$). The attribution rules are written as Haskell-like expressions.

We extend this Ag notation with two new constructs: **Symbols** and **Functions**. The former defines the set of *generic symbols* used by the GenAg. The latter defines the set of *generic semantic functions* used by the GenAg. For each generic symbol we define also its set of inherited and synthesized attributes.

Consider a generic attribute grammar $\mathbf{AG_1}$ rooted R. Suppose that this grammar has a generic symbol $\underline{\mathbf{X}} \in \mathcal{G}$, with $A_{gen}(\underline{\mathbf{X}}) = \{inh1, syn1, syn2\}$ that is partitioned into the set of inherited attributes $A_{inh}(\underline{\mathbf{X}}) = \{inh1\}$ and the set of synthesized attributes $A_{syn}(\underline{\mathbf{X}}) = \{syn1, syn2\}$. We use the following notation to denote this set:

$$\mathbf{Symbols} \quad = \quad \{ \ \underline{\mathbf{X}} < \downarrow inh1, \uparrow syn1, \uparrow syn2 > \quad \}$$

Consider also that $\mathbf{AG_1}$ has a generic function $f \in \mathcal{F}$ that is used to define an occurrence of attribute *syn1*. This function has to be defined in the **Functions** contructor. The (polymorphic) type of every generic semantic function is also defined. The set of generic semantic functions is denoted as follows:

$$\mathbf{Functions} \quad = \quad \{ \quad syn1 \quad : \quad \{ \quad f \quad :: \quad a \ \rightarrow b \rightarrow c \ \} \\ \}$$

Next we present the attribution rules of this generic attribute grammar. The generic symbol $\underline{\mathbf{X}}$ occurs in the productions and in the semantic equations of the GenAg as a “normal” grammar symbol. The same holds for the generic function f which is used as a “normal” semantic function too.

$$
\begin{array}{lcl}
R & & <\uparrow syn1> \\
R & \rightarrow & \textsc{ProdR}\ (\underline{\mathbf{X}}\ \texttt{':'}\ S) \\
 & & \underline{\mathbf{X}}.inh1 \;=\; S.syn1 \\
 & & S.inh1 \;=\; \underline{\mathbf{X}}.syn2 \\
 & & R.syn1 \;=\; \underline{\mathbf{X}}.syn1
\end{array}
\qquad
\begin{array}{lcl}
S & & <\downarrow inh1, \uparrow syn1> \\
S & \rightarrow & \textsc{ProdS}\ (n) \\
 & & S.syn1 \;=\; f\ n\ S.inh1
\end{array}
$$

Observe that the inherited and synthesized attributes of a generic symbol can be derived from the attribute equations, exactly as for normal non-terminal symbols. We define the set of inherited and synthesized attributes in the GenAg specifications to increase their readability.

For presenting generic attribute grammar specifications we introduce a simple desk calculator language Desk. Desk was presented in a recent survey on attribute grammars [Paa95] where it is analysed in great detail. A program in Desk as the following form:

`PRINT` <*Expression*> `WHERE` <*Definitions*>

where <*Expression*> is an arithmetic expression over numbers and defined constants, and <*Definitions*> is a sequence of constant definitions of the form:

<*Constant Name*> `=` <*Number*> `:` <*Type*>

each named constant used in <*Expression*> must be defined in <*Definitions*>, and <*Definitions*> may not give multiple values for a constant. We extend the original language with types, that is, a constant has a type `int` or `real`. The dynamic meaning of a Desk program is defined implicitly as a mapping into a lower-level code. A concrete sentence in Desk and the respective code generated looks as follows:

```
PRINT  1 + x - y              LOADi   1
WHERE  x = 2 : int,           ADDi    2      (x)
       y = 3 : real           SUBr    3      (y)
                              PRINT   0
                              HALT    0
```

This language introduces several typical tasks which are common in real programming languages: *name analysis*, *type checking*, *code generation*. Observe also that it is a one pass right-to-left language because named entities can be used before declaration.

The Desk compiler will be defined as a set of GenAg components. We start by defining a component which performs the static semantics of Desk. The code generation component is defined in section 4.2.

Let us assume that the <*Expression*> part of Desk has one inherited attribute, the environment (*env*), and synthesizes three attributes: the list of identifiers used and not contained in the inherited environment (attribute *errs*), a typed tree for the expression (*tt*) and the inferred type (*type*). Suppose that in a library of GenAg components we have a generic component describing the processing of such expression trees. Therefore we refer to the <*Expression*> part as a generic symbol as follows:

$$\textbf{Symbols} \;=\; \{\ \underline{\mathbf{Exp}} <\downarrow env, \uparrow errs, \uparrow type, \uparrow tt> \ \}$$

and we concentrate on defining the rest of the GenAg. This grammar is presented next. The start symbol is *Desk*.

$$
\begin{array}{lcl}
Desk & & <\uparrow errs> \\
Desk & \rightarrow & \textsc{RootProd}\ (\texttt{'PRINT'}\ \boxed{\underline{\mathbf{Exp}}}\ \texttt{'WHERE'}\ Defs) \\
 & & \underline{\mathbf{Exp}}.env \;=\; Defs.env \\
 & & Desk.errs \;=\; concat\ \underline{\mathbf{Exp}}.errs\ Defs.errs
\end{array}
$$

$$
\begin{array}{lcllcl}
\mathit{Defs} & & \multicolumn{4}{l}{<\uparrow \mathit{env}, \uparrow \mathit{errs}>} \\
\mathit{Defs} & \rightarrow & \multicolumn{4}{l}{\textsc{OneDef}\ (\mathit{Def})} \\
 & & & \mathit{Defs.errs} & = & \mathit{Def.errs} \\
 & & & \mathit{Defs.env} & = & \mathit{Def.env} \\
 & \mid & \multicolumn{4}{l}{\textsc{NoDefs}\ ()} \\
 & & & \mathit{Defs.errs} & = & \mathit{nil} \\
 & & & \mathit{Defs.env} & = & \texttt{[]} \\
\mathit{CD} & & \multicolumn{4}{l}{<\uparrow \mathit{name}, \uparrow \mathit{entry}>} \\
\mathit{CD} & \rightarrow & \multicolumn{4}{l}{\textsc{Assign}\ (\texttt{name '=' num ':' type})} \\
 & & & \mathit{CD.name} & = & \texttt{name} \\
 & & & \mathit{CD.entry} & = & (\texttt{num}, \texttt{type})
\end{array}
$$

$$
\begin{array}{lcllcl}
\mathit{Def} & & \multicolumn{4}{l}{<\uparrow \mathit{env}, \uparrow \mathit{errs}>} \\
\mathit{Def} & \rightarrow & \multicolumn{4}{l}{\textsc{LstDefs}\ (\mathit{Def}\ \texttt{','}\ \mathit{CD})} \\
 & & & \mathit{Def.errs} & = & \texttt{if}\ \mathit{isin}\ (\mathit{CD.name}, \mathit{Def}_2.\mathit{env}) \\
 & & & & & \texttt{then}\ \ \mathit{cons}\ \ \mathit{CD.name}\ \mathit{Def}_2.\mathit{errs} \\
 & & & & & \texttt{else}\ \ \mathit{Def}_2.\mathit{errs} \\
 & & & \mathit{Def.env} & = & (\mathit{CD.name}, \mathit{CD.entry}) : \mathit{Def}_2.\mathit{env} \\
 & \mid & \multicolumn{4}{l}{\textsc{Defi}\ (\mathit{CD})} \\
 & & & \mathit{Def.errs} & = & \mathit{nil} \\
 & & & \mathit{Def.env} & = & \texttt{[}(\mathit{CD.name}, \mathit{CD.entry})\texttt{]}
\end{array}
$$

The generic symbol and its attribute occurrences are used in the semantic equations of the GenAg as normal symbols and attributes. The semantic functions used to define the attribute *env* are part of this GenAg component, although they are not presented in the above grammar. These functions are the (built-in) list functions: the list and the empty list constructor, denoted by `:` and `[]` respectively, and the function *isin* that is the list membership predicate.

The *meaning* of this GenAg component is the attribute *errs*: the only synthesized attribute of the root symbol. This attribute represents the list of errors which occur in a Desk program. This attribute can be a list data structure, that contains the errors, and that might be used by another GenAg component for further processing, or it can be a string representing a pretty printed list of errors that is shown to the user. In order to make possible to (re)use this GenAg component in all these cases we define the semantic functions used to compute attribute *errs* as generic functions.

As we will see later these generic semantic functions can be instantiated with the (built-in) list constructor functions and the meaning of this component is a list of errors. However, those functions can also be instantiated with pretty printing functions and the meaning of the GenAg is a string. Those functions must follow the polymorphic type of the generic semantic functions. This generic functions are declared as follows:

$$
\textbf{Functions} = \{\ \mathit{errs}: \ \{\ \begin{array}{lcl} \mathit{concat} & :: & a \rightarrow a \rightarrow a \\ ,\ \mathit{cons} & :: & b \rightarrow a \rightarrow a \\ ,\ \mathit{nil} & :: & \rightarrow a \end{array} \ \} \ \}
$$

2.2. Circularities

Generic attribute grammars are executable, that is, *efficient and generic implementations* can be automatically derived from generic attribute grammars. In order to derive such implementations, the generic attribute grammars have to guarantee that all the attribute instances of a generic abstract syntax tree are effectively computable. That is, an *order* to compute such attribute instances must exist. Several methods exist that find such an order for "normal" attribute grammars [Kas80, Alb91, Pen94].

Most of the algorithms that analyse attribute grammars for attribute dependencies can be used to handle generic attribute grammars as well. The key idea is to provide those algorithms with the dependencies between the inherited and the synthesized attributes of the generic symbols. Such dependencies cannot be inferred from the GenAg since the attribute equations defining the synthesized attributes of the generic symbols are not included in the GenAg.

In this paper we will present a strict and purely functional implementation for generic attribute grammars. This implementation is based on the visit-sequence paradigm [Kas80] and is restricted to the class of *partitionable attribute grammars* [Alb91].

A straightforward strategy to approximate the dependencies of the generic symbols is to look at the evaluators of these symbols as functions from inherited to synthesized attributes. That is, every synthesized attribute of a generic symbol depends on all the inherited attributes of that same symbol. Consequently, every generic symbol $\mathbf{G} \in \mathcal{G}$ has one single partition:

$$\pi_1(\mathbf{G}) \quad = \quad ([A_{inh}(\mathbf{G})], [A_{syn}(\mathbf{G})])$$

Such an approach, however, may easily introduce fake cyclic dependencies. Suppose that in a production of a GenAg, where a generic symbol $\mathbf{G}$ occurs, one inherited attribute a of $\mathbf{G}$ (transitively) depends on one synthesized attribute b of $\mathbf{G}$. This type of dependencies induces a cyclic dependency since attribute b directly depends on attribute a (recall that in a generic symbol, every synthesized attribute depends on all the inherited ones). In other words, multiple traversal generic symbols are not guaranteed to pass this simplistic approximation.

Consider the GenAg $\mathbf{AG_1}$ defined in section 2.1. Figure 1 shows the dependency graph of production ProdR pasted with the functional dependencies induced by the generic symbol $\mathbf{X}$ and the non-terminal symbol S.

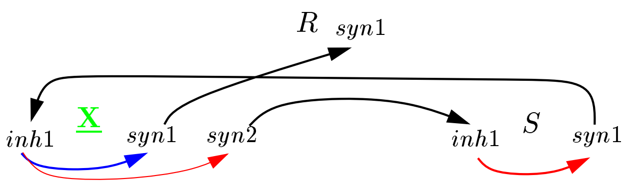

Figure 1: Cyclic dependency graph induced by the functional dependencies of $\mathbf{X}$.

2.2.1. Flow Types

A better strategy to provide the circularity test algorithm with the dependencies between inherited and synthesized attributes of the generic symbols is to explicitly specify such dependencies in the generic attribute grammar. That is, the GenAg defines the *computational pattern* of the generic symbols, the so-called *flow types*. These types define how the information flows from the inherited to the synthesized attributes of every generic symbol.

A flow type of a generic symbol $\mathbf{G}$ is a finite list of *pairs*, each of which defining a *computation pattern* for $\mathbf{G}$. A computation pattern consists of a finite list of inherited attributes of $\mathbf{G}$ (the first element of the pair) and a non-empty finite list of synthesized attributes of $\mathbf{G}$ (the second element of the pair). Thus, every computation pattern defines a function from the inherited to the synthesized attributes of a generic symbol.

Consider the generic symbol $\mathbf{G} \in \mathcal{G}$. A flow type for $\mathbf{G}$ is defined as follows:

$$[\ ([arg_1], [res_1]), \ldots, ([arg_n], [res_n])\ ]$$

, with $arg_1 \cup \cdots \cup arg_n = A_{inh}(\mathbf{G})$ and $res_1 \cup \cdots \cup res_n = A_{syn}(\mathbf{G})$ and $res_1 \cap \cdots \cap res_n = \emptyset$. That is, inherited attributes may be used in different computation patterns, but a synthesized attribute is defined in exactly one. We annotate the generic symbols with the flow types as follows:

$$\begin{array}{llllll} \mathbf{G} & :: & (& (arg_1) & \rightarrow & (res_1) \\ & & , & \cdots & & \\ & & , & (arg_n) & \rightarrow & (res_n) \\ & &) & & & \end{array}$$

, where $arg_j = inh_1, \ldots, inh_k \in A_{inh}(\mathbf{G})$, with $1 \leq j \leq n$.

Let us return to the generic attribute grammar $\mathbf{AG_1}$. Observe that according to the attribution rules of production RootR, the generic symbol **G** is a two traversal symbol: The two traversals are due to the dependency from a synthesized (syn_2) to an inherited attribute (inh_1) of **G**. So, let us define a flow type which specifies a two traversal computation pattern. The first traversal synthesizes attribute syn_2 and the second one uses attribute inh_1. A possible flow type looks as follows:

$$\textbf{Symbols} \;=\; \{ \;\; \mathbf{X} \;\; :: \;\; (\rightarrow syn2, inh1 \rightarrow syn1) \;\; \}$$

This flow type defines the following partitions: $\pi_1(\mathbf{X}) = ([], [syn2])$ and $\pi_2(\mathbf{X}) = ([inh1], [syn1])$. Figure 2 shows the induced dependency graph for production RootR.

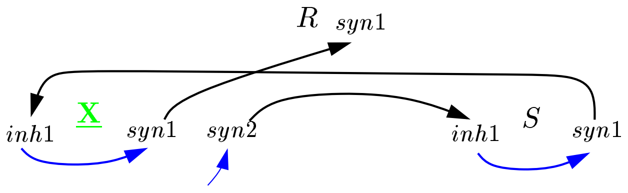

Figure 2: Dependency Graph induced by the flow type of **X**

In section 4.4 we will introduce *modular and generic attribute grammars* and flow types are automatically inferred and do not have to be explicitly specified.

Let us return to the Desk calculator example. The flow type defined for the generic symbol **Exp** is:

$$\textbf{Symbols} \;=\; \{ \;\; \mathbf{Exp} \;\; :: \;\; (\; env \rightarrow (errs, type, tt) \;) \;\; \}$$

Observe that this flow type does not introduce any circularity in the GenAg.

3. Generic Attribute Evaluators

This section presents *Generic Attribute Evaluators* (GenAe), an efficient implementation for *generic attribute grammars*. The generic attribute evaluators are based on purely functional attribute evaluation techniques [KS87, Joh87, Aug93, SS99]. The generic attribute evaluators, however, have to provide a mechanism to handle the generic symbols and semantic functions of the GenAgs.

Next we present strict λ-attribute evaluators that are an efficient functional implementation for partitionable attribute grammars.

3.1. λ-Attribute Evaluators

The strict λ-attribute evaluators consist of a set of *partial parameterized functions*, each performing the computations of one traversal of the evaluator. Those functions, the so-called *deforestated visit-functions*, return as one of their results the visit-functions for the next traversal. Performing the visit corresponds to *totally parameterizing* the deforestated visit-functions and, once again returning the function for the next traversal. The main idea is that for each visit-sub-sequence we construct a function that, besides mapping inherited to synthesized attributes, also returns the function representing the next visit. Any state information needed in future visits is preserved by partially parameterizing a more general function. The only exception is the final deforestated visit-function that returns synthesized attributes only .

Consider the following simplified visit-(sub-)sequences for production $X \rightarrow \textsc{Prod}\ (Y\ Z)$:

$$\begin{array}{ll}
\textbf{plan}\ \textsc{Prod} & \\
\textbf{begin}\ 1 & \texttt{inh}(X.inh_1) \\
\quad \texttt{visit} & (Y,1) \\
\quad \texttt{eval} & \cdots \\
 & \texttt{uses}(X.inh_1,\cdots), \\
\quad \texttt{visit} & (Y,2) \\
\quad \texttt{eval} & (X.syn_1) \\
 & \texttt{uses}(\cdots), \\
\textbf{end}\ 1 & \texttt{syn}(X.syn_1)
\end{array}
\qquad
\begin{array}{ll}
\textbf{begin}\ 2 & \texttt{inh}(X.inh_2) \\
\quad \texttt{visit} & (Z,1) \\
\quad \texttt{eval} & (X.syn_2) \\
 & \texttt{uses}(X.inh_1,\cdots) \\
\textbf{end}\ 2 & \texttt{syn}(X.syn_2)
\end{array}$$

Observe that the inherited attribute $X.inh_1$ must be explicitly passed from the first visit of X (where it is defined) to the second one (where it is used). The non-terminal Y is visited twice, both visits are performed in the first visit to X. These two visit-sequences are implemented by the following two deforestated visit-functions:

$$\begin{array}{ll}
\lambda_{Prod^1} & \lambda_{Y^1}\ \lambda_{Z^1}\ \boxed{inh_1} = ((\lambda_{Prod^2}\ \boxed{inh_1}\ \lambda_{Z^1}), syn_1) \\
\quad \texttt{where} & (\boxed{\lambda_{Y^2}}, \ldots) = \lambda_{Y^1}\ \ldots \\
 & (\ldots) = \boxed{\lambda_{Y^2}}\ \ldots \\
 & syn_1 = \cdots \\
 & \\
\lambda_{Prod^2} & \boxed{inh_1}\ \lambda_{Z^1}\ inh_2 = (syn_2) \\
\quad \texttt{where} & (\ldots) = \lambda_{Z^1}\ \ldots \\
 & syn_2 = f(\boxed{inh_1}, \ldots)
\end{array}$$

$\boxed{inh_1}$ defined in λ_{Prod^1}
used in λ_{Prod^2}

$\boxed{\lambda_{Y^2}}$ partial parameterized in the first traversal and totally parameterized in the second one.

The functions λ_{Y^1} and λ_{Z^1} define the computations for the first traversal of non-terminal symbols Y and Z. The attribute occurrence $X.inh_1$ is passed from the first to the second traversal as a hidden result of λ_{Prod^1} in the form of an extra argument to λ_{Prod^2}. Note also that no reference to visits for non-terminal symbol Y is included in λ_{Prod^2} since all the visits to Y occur during the first visit to P only.

The λ-attribute evaluators are more generic than conventional attribute evaluators since their interface specification does not contain any explicit references to data types representing the abstract syntax trees. Attribute instances needed in different traversals of the evaluator are passed between traversals as results/arguments of partial parameterized visit-functions. No additional data structures are required to handle them, like trees [Kas91b, PSV92, SKS97] or stacks and queues [AS91]. The visit-functions find *all the values* they need in their arguments. Furthermore, the resulting evaluators are higher-order attribute evaluators. The arguments of the evaluators visit-functions are other visit-functions. Finally, the evaluators have an efficient memory usage: data not needed anymore is no longer referenced. References to grammar symbols and attribute instances can efficiently be reclaimed as soon as they have played their semantic role.

Parse-time attribute evaluation is achieved as a by-product of our techniques: the deforestated visit-functions are directly called during parsing. No syntax tree is explicitly constructed. A typical parser fragment (derived from $\mathbf{AG_1}$, in this case) looks as follows[1]:

[1] We use Happy [Mar97], an Yacc equivalent for Haskell.

$$
\begin{array}{lll}
R & : & X \;\; \texttt{':'} \;\; S \\
  & \{ & \lambda_{ProdR^1} \;\; \$1 \;\; \$3 \;\; \} \\
S & : & n \\
  & \{ & \lambda_{ProdS^1} \;\; \$1 \;\; \}
\end{array}
$$

, where λ_{ProdR^1} and λ_{ProdS^1} are calls to the deforestated visit-function derived from $\mathbf{AG_1}$. These functions are presented in section 3.2.1.

3.2. Deriving Generic Attribute Evaluators

We define a mapping from generic attribute grammars into generic attribute evaluators. This mapping handles the generic symbols and semantic functions in the "*standard*" way: they are extra parameters of the λ-attribute evaluators. That is, for every generic symbol the attribute evaluator receives an extra argument corresponding to the evaluator of that symbol. The generic semantic functions are also handled as extra arguments to the generic evaluator. In other words, the λ-attribute evaluator derived for a generic attribute grammar *gag* is parameterized with the *static parameters* $\mathcal{P}$ of *gag*.

This technique is orthogonal to the strict and lazy implementation of attribute grammars presented in [Sar]. Thus, strict and lazy implementations can be derived for generic attribute grammars. The examples of generic attribute evaluators presented in this section are based on the strict implementation only.

3.2.1. Generic Symbols

The generic symbols are handled as normal non-terminal symbols: the visit-functions derived for the productions, where a generic symbol occurs, receive as an extra argument the generic evaluator which decorates the generic symbol. This is just as normal grammar symbols are handled. The generic attribute evaluators, however, refer to the visit-functions which decorate the generic symbols as parameters of the GenAe.

In order to derive strict λ-attribute evaluators for generic attribute grammars the attribute evaluation order must first be statically computed. Traditional attribute grammar scheduling algorithms can be used to compute such an evaluation order, provided that the dependencies between the inherited and synthesized attributes of the generic symbols are known. That is, the Ag scheduling algorithms have to *know* the partitions of the generic symbols, like for the circularity test of generic attribute grammars. Thus, the flow types are used to provide the scheduling algorithm with the evaluation order of the generic symbols.

Consider the generic attribute grammar $\mathbf{AG_1}$ and the flow type of $\underline{\mathbf{X}}$ as defined in section 2.2.1. According to the partitions induced by the flow type and the dependencies defined in both productions of $\mathbf{AG_1}$ (see figure 2) the visit-sequences produced by the the scheduling algorithm are:

```
plan ProdR
begin 1    inh()
   visit   (X, 1)
   eval    (S.inh1)
   visit   (S, 1)
   eval    (X.inh1)
   visit   (X, 2)
   eval    (R.syn1)
end 1      syn(R.syn1)
```

```
plan ProdS
begin 1    inh(S.inh1)
   eval    (S.syn1)
end 1      syn(S.syn1)
```

The visit-sequences can be directly implemented as λ-attribute evaluators using the techniques described in [SS99]. For the sake of completeness the resulting deforestated visit-functions are shown next. The generic semantic function f is considered in this example as a normal semantic function. Generic functionas are discussed in next section.

$$\begin{array}{lll} \lambda_{ProdR^1} & \boxed{\lambda_{\mathbf{X}^1}}\ \lambda_{S^1} = (syn1) \\ \texttt{where} & (\lambda_{\mathbf{X}^2}, syn2) = \boxed{\lambda_{\mathbf{X}^1}} \\ & (syn1_2) = \lambda_{S^1}\ \ syn2 \\ & (syn1) = \lambda_{\mathbf{X}^2}\ \ syn1_2 \end{array} \qquad \begin{array}{ll} \lambda_{ProdS^1} & \mathtt{n}\ \ inh1 = (syn1) \\ \texttt{where} & (syn1) = f\ \ \mathtt{n}\ \ inh1 \end{array}$$

As expected the evaluator of the generic symbol $\mathbf{X}$ is one argument of the visit-function λ_{ProdR^1}. Observe that that evaluator represents the first visit to the generic symbol only. When such a function is applied to the inherited attributes of the first visit (an empty set, in this case), it returns the visit-function to the next visit: Exactly as "*normal*" deforestated visit-functions of λ-attribute evaluators.

Consider the productions RootProd and Assign of the Desk calculator. The deforestated visit-functions derived for these productions are:

$$\begin{array}{ll} \lambda_{RootProd^1} & \boxed{\lambda_{Exp^1}}\ \lambda_{Def^1} = (errs) \\ \texttt{where} & (env_3, errs_3) = \lambda_{Def^1} \\ & (errs_2, type_2, tt_2) = \boxed{\lambda_{Exp^1}}\ env_3 \\ & (errs) = concat\ \ errs_2\ errs_3 \end{array} \qquad \begin{array}{ll} \lambda_{Assign^1} & \lambda_{name^1}\ \lambda_{num^1}\ \lambda_{type^1} = (name, entry) \\ \texttt{where} & (name) = \lambda_{name^1} \\ & (entry) = (\lambda_{num^1}, \lambda_{type^1}) \end{array}$$

, where $\boxed{\lambda_{Exp^1}}$ is the reference to the partial parameterized visit-function that is obtained when parsing the expression part of a Desk program. In the body of visit-function $\lambda_{RootProd^1}$ we can see the right to left pass of the attribute evaluator: first the function that represents the evaluation of the <*Definitions*> part of Desk is called (and it returns the environment). After that, the function which represents the evaluation of the <*Expression*> part is computed (and it uses the computed environment). Note also that the deforestated visit-function λ_{Def^1} has all the parameters it needs at parse-time (no parameters in this case). Thus, the calls to that function are totally parameterized at parse-time. Consequently, the <*Definitions*> part of Desk is decorated during parsing.

The arguments of the visit-function λ_{Assign^1} are the tokens produced by the "external" lexical analyser. The deforestated visit-functions are completely generic as shown by their type definitions:

$$\begin{array}{lll} \lambda_{RootProd^1} & :: & (a \rightarrow (b, c, d)) \rightarrow (a, b) \rightarrow b \\ \lambda_{Assign^1} & :: & a \rightarrow b \rightarrow c \rightarrow (a, (b, c)) \end{array}$$

In this evaluator and in the GenAg, for example, nothing is defined about the type of the identifiers of the language. They can be a sequence of characters, a single one or even a numeral. The GenAg and the evaluator can be reused in all those cases.

3.2.2. Generic Semantic Functions

The generic semantic functions are handled as extra arguments of the evaluators. We consider two possibilities: the generic semantic functions are normal *inherited attributes*, or the generic functions are *syntactic referenced symbols*. In the next sections we describe both approaches.

3.2.3. Generic Semantic Functions as Inherited Attributes

They are *inherited attributes* of the root symbols and they are passed down in the tree to the nodes where they are used as normal attributes. Furthermore, the attribute grammar scheduling algorithms are used in the standard way to schedule their "computation", because the generic semantic functions have become normal attributes. In this section we show how to transform a GenAg wich contains generic functions into an equivalent one without generic functions. As a result of this transformation, we can use the techniques described thus far to derive generic attribute evaluators.

A generic attribute grammar with generic functions can be transformed into an equivalent grammar with no generic functions as follows: Let $gag = (gg, A, E, \mathcal{F})$ be a generic attribute grammar and R be the root of the generic context free grammar gg.

1. For every generic semantic function $f \in \mathcal{F}$ we add an inherited attribute $\downarrow f$ to $A_{inh}(R)$.
2. Every occurrence of f in E is replaced by $\uparrow \mathrm{R}.f$.

, where $\uparrow \mathrm{X}.f$ denotes the access to a remote attribute of an *ascendent* non-terminal symbol[2].

Consider the GenAg $\mathbf{AG_1}$. This grammar is transformed into the following one:

$$
\begin{array}{lll}
R & & <\boxed{\downarrow f}, \uparrow syn1> \\
R & \rightarrow & \mathrm{ProdR}\ (\underline{\mathbf{X}}\ \texttt{':'}\ S) \\
 & & \quad \underline{\mathbf{X}}.inh1 \;=\; S.syn1 \\
 & & \quad S.inh1 \;=\; \underline{\mathbf{X}}.syn2 \\
 & & \quad R.syn1 \;=\; \underline{\mathbf{X}}.syn1
\end{array}
\qquad
\begin{array}{lll}
S & & <\downarrow inh1, \uparrow syn1> \\
S & \rightarrow & \mathrm{ProdS}\ (n) \\
 & & \quad S.syn1 \;=\; \boxed{\uparrow \mathrm{R}.f}\ \ n\ \ S.inh1
\end{array}
$$

, and the derived λ-attribute evaluator looks as follows:

$$
\begin{array}{ll}
\lambda_{ProdR^1} & \lambda_{\underline{\mathbf{X}}^1}\ \lambda_{S^1}\ \boxed{f} = (syn1) \\
\texttt{where} & (\lambda_{\underline{\mathbf{X}}^2}, syn2) = \boxed{\lambda_{\underline{\mathbf{X}}^1}} \\
 & (syn1_2) = \lambda_{S^1}\ \ syn2\ \boxed{f} \\
 & (syn1) = \lambda_{\underline{\mathbf{X}}^2}\ \ syn1_2
\end{array}
\qquad
\begin{array}{ll}
\lambda_{ProdS^1} & \texttt{n}\ \ inh1\ \boxed{f} = (syn1) \\
\texttt{where} & (syn1) = \boxed{f}\ \ \texttt{n}\ \ inh1
\end{array}
$$

This technique, however, introduces type conflicts in the GenAe when the productions of a GenAg are extended. This type conflict will be explained in section 4.1.

3.2.4. Generic Semantic Functions as Syntactic Symbols

This section introduces a second approach to handle generic semantic functions: they are *syntactically referenced symbols*. That is, each visit-function defining the first visit to a particular constructor receives an extra argument for every generic function used in the respective production. The generic semantic functions are passed to the visits where they are applied, as arguments/results of the visit-functions which perform the different visits to that particular constructor. This is exactly as *syntactically referenced symbols* (*i.e.* grammar symbols which are used as normal values in the semantic equations of a Higher-order Attribute Grammar [VSK89]) are handled by the λ-attribute evaluators. Thus, the techniques used to handle syntactic references can be used to handle generic symbols too. In this paper we present the evaluators obtained with this technique only. For a formal derivation of such evaluators the reader is referred to [Sar].

Consider the GenAg $\mathbf{AG_1}$ again. The generic attribute evaluator obtained with this technique is:

[2] We use the notation of the Ssl language [RT89].

$$
\begin{array}{lll}
\lambda_{ProdR^1} & \lambda_{\mathbf{X}^1}\ \lambda_{S^1} = (syn1) & \\
\texttt{where} & (\lambda_{\mathbf{X}^2}, syn2) = \boxed{\lambda_{\mathbf{X}^1}} & \\
 & (syn1_2) = \lambda_{S^1}\ \ syn2 & \\
 & (syn1) = \lambda_{\mathbf{X}^2}\ \ syn1_2 &
\end{array}
\qquad
\begin{array}{ll}
\lambda_{ProdS^1} & \boxed{f}\ \mathtt{n}\ \ inh1 = (syn1) \\
\texttt{where} & (syn1) = \boxed{f}\ \ \mathtt{n}\ \ inh1
\end{array}
$$

Observe that the visit-function call λ_{S^1} in the body of λ_{ProdR^1} is partial parameterized with the generic function f at parse-time. That function call corresponds to the total parameterization of the visit-function.

$$
\begin{array}{lll}
R & : & X\ \texttt{':'}\ S \\
 & \{ & \lambda_{ProdR^1}\ \ \$1\ \ \$3\ \ \} \\
S & : & n \\
 & \{ & \lambda_{ProdS^1}\ \ f\ \ \$1\ \ \}
\end{array}
$$

The generic functions are now the first arguments of the visit-functions. This approach has an important advantage when compared to the previous one: the visit-functions can be partially evaluated [JGS93] with respect to generic semantic functions. Note that they are the partial parameters of the GENAG. That is, when we parameterize the generic function of the GENAG with a particular function (that is a static parameter of the GENAG) we also partially evaluate the respective evaluator with respect to that function. As a result of partial evaluation the evaluator is a "standard" λ-attribute evaluator: all the semantic functions are part of the evaluator and they are not arguments of the visit-functions.

Let us consider the DESK calculator again and present a GENAG component which defines the processing of expression trees. The root symbol is *Exp*.

$$
\begin{array}{llllllll}
\textbf{Functions} & = & \{ & tt: & \{ & add & :: & a \to b \to c \to c \\
 & & & & , & fac & :: & a \to b \to c\ \} \\
 & & , & errs: & \{ & concat & :: & d \to d \to d \\
 & & & & , & cons & :: & e \to d \to d \\
 & & & & , & nil & :: & \to d\ \ \} \\
 & & \} & & & & &
\end{array}
$$

$$
\begin{array}{llll}
Exp & & \multicolumn{2}{l}{<\downarrow env, \uparrow errs, \uparrow type, \uparrow tt>} \\
Exp & \to & \multicolumn{2}{l}{\textsc{Add}\ (Exp\ \texttt{'+'}\ Fac)} \\
 & & Exp_2.env = & Exp.env \\
 & & Fac.env = & Exp.env \\
 & & Exp.errs = & \boxed{concat}\ \ Exp_2.errs\ \ Fac.errs \\
 & & Exp.type = & inftype\ Exp_2.type\ Fac.type \\
 & & Exp.tt = & \boxed{add}\ \ Fac.val\ Exp.type\ Exp_2.tt \\
 & | & \multicolumn{2}{l}{\textsc{Factor}\ (Fac)} \\
 & & Fac.env = & Exp.env \\
 & & Exp.errs = & Fac.errs \\
 & & Exp.type = & Fac.type \\
 & & Exp.tt = & fac\ Fac.val\ Exp.type
\end{array}
\qquad
\begin{array}{llll}
Fac & & \multicolumn{2}{l}{<\downarrow env, \uparrow errs, \uparrow type, \uparrow val>} \\
Fac & \to & \multicolumn{2}{l}{\textsc{Var}\ (\mathtt{name})} \\
 & & Fac.val = & getval\ \mathtt{name}\ Fac.env \\
 & & Fac.type = & gettype\ Fac.val \\
 & & Fac.errs = & \ \texttt{if}\ isin\ \mathtt{name}\ Fac.env \\
 & & & \ \texttt{then}\ \ nil \\
 & & & \ \texttt{else}\ \ cons\ \ \mathtt{name}\ \ nil \\
 & | & \multicolumn{2}{l}{\textsc{Number}\ (\mathtt{num})} \\
 & & Fac.val = & \mathtt{num} \\
 & & Fac.type = & gettype\ Fac.val \\
 & & Fac.errs = & nil
\end{array}
$$

The semantic functions used to compute the value of synthesized attributes *errs* and the typed tree *tt* are defined as generic functions. The other semantic functions (*e.g.* *getval*, *gettype* , etc) are normal (static) semantic functions that are part of this GENAG. The single deforestated visit-function derived for the production ADD is presented next:

$$
\begin{array}{ll}
\lambda_{Add^1} & \boxed{concat}\ \boxed{add}\ \lambda_{Exp_2^1}\ \lambda_{Fac^1}\ env = (errs, type, tt) \\
\texttt{where} & (errs_3, type_3, val_3) = \lambda_{Fac^1}\ \ env \\
 & (errs_2, type_2, tt_2) = \lambda_{Exp_2^1}\ \ env \\
 & (errs) = \boxed{concat}\ \ errs_2\ \ errs_3 \\
 & (type) = inftype\ type_2\ type_3 \\
 & (tt) = \boxed{add}\ val_3\ type\ tt_2
\end{array}
$$

We show next the (complicated) type definition for this deforestated visit-function. This function and its type will be used in the next section to explain the composition of GenAg components.

$$
\begin{array}{lll}
\lambda_{Add^1} & :: & (a \rightarrow b \rightarrow c) \rightarrow (d \rightarrow Type \rightarrow e \rightarrow f) \rightarrow (g \rightarrow (a, Type, e)) \rightarrow (g \rightarrow (b, Type, d)) \rightarrow g \\
 & & \rightarrow (c, Type, f)
\end{array}
$$

As we can see in this type definition, the first four type sub-expressions represent the polymorphic types of four functions: the two generic functions and the two visit-functions (derived for the right-hand side non-terminal symbols). The type variable g represents the type of the environment. The visit-function returns one triple, where c represents the type of the list of errors, $Type$ is the derived type for the expression (inferred from the static semantic functions) and f is the type variable that represents the type of the typed tree.

4. Semantic Compositionality

This section discusses the semantic compositionality of generic attribute grammars.

4.1. Extending the Productions of a Non-terminal

The generic attribute evaluators are independent of the abstract tree data type. This data type freeness allows the GenAg and the GenAe to be easily extended with new productions. Suppose we want to extend a generic attribute grammar with new productions. In a traditional Ag implementation, the existing attribute evaluator will have to be modified, since the type of the abstract syntax tree changes. As a result the attribute evaluator for the complete Ag will have to be produced again.

In our implementation, however, we can define a new GenAg component where the new production and its semantic equations are specified. This component can be analysed separately and the deforestated visit-function(s) for the production can be produced. No "global" tree data type has to be modified. In order to compute the order of the attribute evaluation we have to provide the scheduling algorithm with the flow type of the non-terminal symbol on the left-hand side of the production.

Let us return to the Desk example. Suppose that we want to extend the productions of the expression GenAg in order to allow the operation $\diamond$, which is not supported by the reused component. Using generic attribute grammars we define this extension in one new component. The non-terminal symbol in the left-hand side of the grammar is defined as a generic symbol and its flow type is specified. The new production and its attribute equations are defined as follows:

$$
\begin{array}{lllllll}
\textbf{Symbols} & = & \{ & \underline{\textbf{Exp}} & :: & (\ env \to (errs, type, tt)\ ) & \} \\
\textbf{Functions} & = & \{ & tt: & \{ & dia :: a \to b \to c \to c & \} \\
 & & , & errs & \{ & concat :: d \to d \to d & \} \\
 & & \} & & & &
\end{array}
$$

$$
\begin{array}{lll}
\underline{\textbf{Exp}} & & <\downarrow env, \uparrow errs, \uparrow type, \uparrow tt > \\
\underline{\textbf{Exp}} & \to & \text{DIA}\ (Exp\ \text{'}\diamond\text{'}\ Fac) \\
 & & Exp_2.env \ = \ Exp.env \\
 & & Fac.env \ = \ Exp.env \\
 & & Exp.errs \ = \ concat\ \ Exp_2.errs\ \ Fac.errs \\
 & & Exp.type \ = \ inftype\ Exp_2.type\ Fac.type \\
 & & Exp.tt \ = \ dia\ \ Fac.val\ Exp.type\ Exp_2.tt
\end{array}
$$

Observe that the start symbol S of this GENAG is the generic symbol **Exp**. This grammar component is used to extend its production only. From this GENAG component a deforestated visit-function is generated. The header of the function derived for this component is:

$$
\lambda_{Dia^1} \quad concat\ \boxed{dia}\ \lambda_{Exp_2^1}\ \lambda_{Fac^1}\ env = (errs, type, tt)
$$

Suppose now that the generic function $\boxed{dia}$ is handled as one inherited attribute, as proposed in the first technique to handle generic semantic functions. In this case, a new inherited attribute *dia* is added to non-terminal symbol *Exp*. As a result all the visit-functions derived for the productions applied on *Exp* have to be extended with an extra argument too. If we consider the separate compilation of the modules, however, we get a type error during evaluation since the visit-functions derived for those productions have different types!

4.2. Composing GENAG Components

The generic attribute grammars are efficiently and easily composed. The generic functions of one GENAG can be instantiated with the visit-functions of the evaluator derived from another GENAG. Because this evaluator is totally deforestated no intermediate data structure is constructed nor traversed. Furthermore, we can give different semantics to a GENAG by instantiating its static parameters (*i.e.* the generic functions and the generic symbols) with different arguments.

Consider the code generation task of the DESK calculator. The code generation is performed according to the typed tree computed by the type checker. The typed tree attribute is one of the synthesized attributes of the expression part of DESK, as defined in section 2.1. In the GENAG for expressions that attribute is defined using generic functions. As a result we can, for example, parameterize the expression GENAG with a tree algebra and explicitly construct the typed tree. This tree can be defined by the following algebraic data type:

$$
\begin{array}{lllll}
\texttt{data} & ETree & = & \text{CONSTREE} & Num\ \ Type\ \ ETree \\
 & & & \text{FACTREE} & Num\ \ Type
\end{array}
$$

, where Num and $Type$ are the types of the correspondent terminal symbols of the GENAG. The constructs of the tree data type follow the types defined for the generic functions. Thus, we can instantiate the generic functions with these constructor functions as follows:

$$
\begin{array}{lll}
add & = & \text{CONSTREE} \\
fac & = & \text{FACTREE}
\end{array}
$$

In this case the evaluation of attribute *tt* actually constructs the typed tree as we can see in the type definition of the resulting deforestated visit-function:

$$
\begin{array}{lll}
\lambda_{Add^1} & :: & (a \to b \to c) \to (d \to Type \to ETree \to ETree) \to (g \to (a, Type, ETree)) \\
 & & \to (g \to (b, Type, d)) \to g \to (c, Type, ETree)
\end{array}
$$

In a traditional AG implementation this tree is traversed later on in order to generate the assembly code. Thus, this tree is the intermediate structure which glues the GENAG components.

Let us now present one GENAG component that describes the code generation task.

$$
\begin{array}{lll}
\mathit{Exp} & & <\uparrow \mathit{code}> \\
\mathit{Exp} & \rightarrow & \textsc{CodeAdd}\ (\texttt{num}\ \texttt{type}\ \mathit{Exp}) \\
 & & \quad \mathit{Exp.code} \ = \ \texttt{if}\ \mathit{isinttype}\ \texttt{type}\ \texttt{then}\ \ \mathit{Exp}_2.\mathit{code}\ \texttt{++}\ [\text{``}\texttt{ADDi}\text{ ``}\ \texttt{++}\ \texttt{num}] \\
 & & \qquad\qquad\qquad \texttt{else}\ \ \mathit{Exp}_2.\mathit{code}\ \texttt{++}\ [\text{``}\texttt{ADDr}\text{ ``}\ \texttt{++}\ \texttt{num}] \\
 & | & \textsc{CodeFac}\ (\texttt{num}\ \texttt{type}) \\
 & & \quad \mathit{Exp.code} \ = \ \texttt{if}\ \mathit{isinttype}\ \texttt{type}\ \texttt{then}\ \ [\text{``}\texttt{LOADi}\text{''}\ \texttt{++}\ \texttt{num}] \\
 & & \qquad\qquad\qquad \texttt{else}\ \ [\text{``}\texttt{LOADr}\text{''}\ \texttt{++}\ \texttt{num}]
\end{array}
$$

, and derive the deforestated generic attribute evaluator. We show next the deforestated visit-function derived for the CODEADD production only:

$$
\begin{array}{l}
\lambda_{CodeAdd^1}\quad \lambda_{num^1}\ \lambda_{type^1}\ \lambda_{Exp^1_2} = (\mathit{code}) \\
\quad \texttt{where}\quad (\mathit{code}_2) = \lambda_{Exp^1_2} \\
\qquad\qquad\quad (\mathit{code}) = \ \texttt{if}\ \mathit{isinttype}\ \lambda_{type^1}\ \texttt{then}\ \ \mathit{code}_2\ \texttt{++}\ [\text{``}\texttt{ADDi}\text{ ``}\ \texttt{++}\ \lambda_{num^1}] \\
\qquad\qquad\qquad\qquad\quad \texttt{else}\ \ \mathit{code}_2\ \texttt{++}\ [\text{``}\texttt{ADDr}\text{ ``}\ \texttt{++}\ \lambda_{num^1}]
\end{array}
$$

Using the deforestated generic attribute evaluators we can instantiate the generic functions with the deforestated visit-functions derived for the code generation GENAG. We compose the GENAG components as follows:

$$
\begin{array}{lcl}
\mathit{add} & = & \lambda_{CodeAdd^1} \\
\mathit{fac} & = & \lambda_{CodeFac^1}
\end{array}
$$

Observe that these visit-functions have all the arguments they need during the decoration of the expressions trees. As result, *the code generation is actually computed during the evaluation of the expression trees.* Furthermore, no intermediate data structure (*e.g.* the typed tree) is constructed. The type definition of the visit-function is:

$$
\begin{array}{lcl}
\lambda_{Add^1} & :: & (a \rightarrow b \rightarrow c) \rightarrow (d \rightarrow Type \rightarrow [String] \rightarrow [String]) \rightarrow (g \rightarrow (a, Type, [String])) \\
 & & \rightarrow (g \rightarrow (b, Type, d)) \rightarrow g \rightarrow (c, Type, [String])
\end{array}
$$

As a result the single visit-function which decorates the expression part of the DESK language returns, as one of its results, a list with the assembly code. The intermediate typed tree is not constructed not traversed. Observe also that the list (with the generated code) can also be eliminated if we define the pretty printing of the assembly code in an attribute grammar setting and derive the correspondent deforestated visit-functions. In this case, the list is eliminated and the previous function returns a string: the assembly code (see [Sar] for details).

4.3. Inter-Module Attribute Dependencies

The main problem in a purely functional implementation of attribute grammars is to handle attribute instances that are computed during one traversal of the evaluator and used in a future one, the so-called *inter-traversal attribute dependencies* [Pen94]. A traditional imperative approach stores such values in the nodes of the tree. The λ-attribute evaluators explicitly pass those values between visits, as normal arguments/results of the visit-functions.

A similar problem occurs with the separate compilation of GENAG components: How do we pass attribute values computed in one GENAG component and used in a different component? We use the approach of the deforestated evaluators: Those values are arguments/results of the visit-functions derived for the GENAG components. No additional data structures are required.

Consider the expression GenAg presented in section 3.2.4. The value of the attribute ***type*** is defined in this GenAg and it is used in the code generation component. The attribute is passed between components in the "standard" way: The generic function which defines the typed tree is partially parameterized with the attribute ***type***. When such generic function is instantiated and totally parameterized the value of the attribute ***type*** is available.

4.4. Modularity Constructs

This section briefly presents a modular notation for generic attribute grammars. A generic attribute grammar component is extended with a *name* and a set of *imported modules*, the so-called *modular and generic attribute grammar*. The presentation in this paper is necessarily short relying entirely on the Desk example. A more detailed and formal presentation appears in [Sar].

A modular and generic attribute grammar extends a GenAg with a module's name $\mathcal{N}$ and a set of imported modules $\mathcal{M}$. Every generic symbol $\underline{\mathbf{X}} \in \mathcal{G}$ of a modular component has to be defined as a (normal) grammar symbol X in one of the imported modules $m \in \mathcal{M}$. We say that a modular and generic Ag is complete if the underlying GenAg is complete and each generic symbol $\underline{\mathbf{X}}$ is also one grammar symbol X of the imported GenAgs. Futhermore, the set of attributes of symbols $\underline{\mathbf{X}}$ and X must be equal. Moreover, the flow type of a generic symbol $\underline{\mathbf{X}}$ is the set of partitions of symbol X. The partitions of the grammar symbol X are computed (using standard Ag techniques) when deriving the evaluator for the GenAg component where X is defined. As a result it is not needed to specify the flow type for the generic symbols: the flow types are the partitions of the grammar symbols which instantiates the generic symbols. We consider that the partitions (*i.e.* the flow types) of the grammar symbols of a modular and generic attribute grammar m are available to the modular grammar which imports m.

In order to write modular and generic attribute grammars we add two new constructs to our specification language: **Module** and **Import**, which define the name of the module and the set of modules imported, respectively. Furthermore, the occurrences of generic symbols in the attribution rules are annotated with the module where the symbol is defined as follows:

<*module name*>_<*generic symbol*>.<*attribute*>

Let us define a modular and GenAg component, called Semantics, which describes the static semantics for the Desk language. This component imports two GenAgs: the Expression GenAg and a GenAg which defines the unparsing of the list of errors. Furthermore, a set of generic semantic functions is defined in order to *glue* the `Semantics` and the `CodeGen` components.

$$
\begin{array}{lll}
\textbf{Module} & \textsc{Semantics} & \\
\textbf{Root} & \mathit{Desk} & \\
\textbf{Import} & = & \{\ \texttt{Errors}\ ,\ \texttt{Expression}\ \} \\
\textbf{Symbols} & = & \{\ \underline{\mathbf{Exp}}\ \} \qquad \texttt{-- no flow type needed} \\
\textbf{Functions} & = & \{\ \mathit{code}: \{\ \mathit{coddesk} :: a \to b\ ,\ \mathit{codadd} :: a \to b \to c \to c \\
& & \qquad\qquad ,\ \mathit{coddia} :: a \to b \to c \to c\ ,\ \mathit{codfac} :: a \to b \to c\ \} \\
& & \}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{Desk} & & <\uparrow \mathit{errs}, \uparrow \mathit{code}> \\
\mathit{Desk} & \to & \textsc{RootProd}\ (\texttt{'PRINT'}\ \underline{\mathbf{Exp}}\ \texttt{'WHERE'}\ \mathit{Defs}) \\
& & \texttt{Expression\_}\underline{\mathbf{Exp}}.\mathit{add} \ = \ \mathit{codadd} \\
& & \texttt{Expression\_}\underline{\mathbf{Exp}}.\mathit{fac} \ = \ \mathit{codfac} \\
& & \texttt{Expression\_}\underline{\mathbf{Exp}}.\mathit{env} \ = \ \mathit{Defs.env} \\
& & \mathit{Desk.errs} \ = \ \texttt{Errors\_}\mathit{Err.concat}\ \texttt{Expression\_}\underline{\mathbf{Exp}}.\mathit{errs}\ \ \mathit{Defs.errs} \\
& & \mathit{Desk.code} \ = \ \mathit{coddesk}\ \texttt{Expression\_}\underline{\mathbf{Exp}}.\mathit{tt}
\end{array}
$$

Observe that the flow type of the generic symbol **Exp** is not defined in the AG. The partitions of *Exp* inferred when producing the attribute evaluator for the `Expression` GENAG are the flow types. Such partitions are available when analysing the `Semantics` AG since it imports the `Expression` GENAG.

Let us show the fragment of the SEMANTICS grammar which defines the processing of the expression tree. When writing a GENAG component the syntax underlying the attribute grammar is reduced to the entities which have a role in the attribute evaluation algorithm. Thus, let us assume that the reused expression GENAG is defined over the abstract syntax of the expression trees instead of over the concrete one (as defined previously). Furthermore, we also extend the expression component with an additional production. This fragment is shown next.

$$
\begin{array}{lcl}
\boxed{\mathbf{Exp}} & & < \downarrow env, \uparrow errs, \uparrow type, \uparrow res > \\
Exp & \rightarrow & \textsc{AddExp}\ (Exp_2\ \texttt{'+'}\ Fac) \\
 & & \quad \texttt{Expression\_}\textsc{Add}\ (Exp_2\ Fac) \\
 & | & \textsc{Factor}\ (Fac) \\
 & & \quad \texttt{Expression\_}\textsc{Factor}\ (Fac) \\
 & | & \textsc{DiaExp}\ (Exp_2\ \texttt{'}\diamond\texttt{'}\ Fac) \\
 & & \quad \begin{array}{lcl}
 Exp_2.env & = & Exp.env \\
 Fac.env & = & Exp.env \\
 Exp.errs & = & \texttt{Errors\_}Err.concat\ \ Exp_2.errs\ Fac.errs \\
 Exp.type & = & checktype\ Exp_2.type\ Fac.type \\
 Exp.res & = & coddia\ \ Fac.val\ Exp.type\ Exp_2.res
 \end{array}
\end{array}
$$

The semantic equations of the productions ADDEXP and FACTOR specify which visit-functions are used to decorate the instances of those productions (*i.e.* the respective subtrees). The new production and its semantic equations are defined as usually.

5. Related Work and Implementation

Several attribute grammar extensions allowing modularity and reusability of attribute grammar components have been developed. Most existing attribute grammar systems, however, use syntactic compositionality: a monolithic AG is first constructed out of modular descriptions. This type of modularity allows the semantic rules in one module to use attributes defined in another module. This allows a separate description of the phases without the need for defining and constructing any glue. This approach has one important drawback: the separate analysis of modules is not supported.

Our work, however, aims at semantic compositionality: the separate analysis and compilation of attribute grammar components. When writing AGs in a component style the interface between two components has to be defined. There are two approaches that can be taken: Define a *special purpose data type* or a *glue* grammar for this specific task. This is, for example, the approach taken in *composable attribute grammars* [FMY92]. In this approach, however, when the interface changes the components have to be recompiled. Furthermore, the "glue" is actually constructed.

The second approach is to define a *generic data type* in which the information may be encoded. Examples of such representations are S-expressions or rose-trees. We have taken this second approach, in which functions are used as the uniform representation: in this way we get the genericity of the second approach. Furthermore, since the functions are totally deforestated no glue is actually constructed.

The generic attribute grammars have been implemented in a very experimental version of the LRC system [KS98]: A purely functional attribute grammar system developed in our department[3]. The LRC processes higher-order attribute grammars, written in a super-set of SSL [RT89], and produces purely

[3] LRC is available at the address: `http://www.cs.uu.nl/people/matthys/lrc_html/`

functional attribute evaluators. Incremental attribute evaluation is achieved via function memoization.

The LRC generates C based attribute evaluators and more recently we have incorporated a new back-end in order to generate HASKELL based evaluators. The λ-attribute evaluators and the generic attribute evaluators are produced in HASKELL only.

The modular constructs for generic attribute grammars are currently being incorporated to LRC. We are also planing to incorporate a type inference system into LRC. In the current implementation of LRC the generic and deforestated attribute evaluators rely on the HASKELL type system to perform the type checking.

6. Conclusions

This paper introduced generic attribute grammars. Generic attribute grammars provide the support for genericity, reusability and modularity in the context of attribute grammars. A generic attribute grammar is an executable component: efficient attribute evaluators are derived for a GENAG. The λ-attribute evaluators are particularly suitable to implement generic attribute grammars since they are completely generic: no data type has to be defined.

The generic attribute evaluators are deforestated and no intermediate data structure is constructed/used to "*glue*" the components of a GENAG system or to pass attribute values between different components. Furthermore, these techniques are orthogonal with the function memoization techniques used to achieve incremental attribute evaluation [PSV92].

Bibliography

[Alb91] Henk Alblas. Attribute evaluation methods. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 48–113. Springer-Verlag, 1991.

[AS91] Rieks Akker and Erik Sluiman. Storage Allocation for Attribute Evaluators using Stacks and Queues. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 140–150. Springer-Verlag, 1991.

[Aug93] Lex Augusteijn. *Functional Programming, Program Transformations and Compiler Construction*. PhD thesis, Eindhoven Technical University, October 1993.

[FMY92] Rodney Farrow, Thomas J. Marlowe, and Daniel M. Yellin. Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation. In *19th ACM Symp. on Principles of Programming Languages*, pages 223–234, Albuquerque, NM, January 1992. ACM press.

[Hen93] Pedro Rangel Henriques. *Attributes and Modularity in the Specification of Formal Languages*. PhD thesis, Departamento de Informática, Universidade do Minho, Portugal, March 1993. (In Portuguese).

[JGS93] Neil Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall Inernational Series in Computer Science. Prentice-Hall, 1993.

[Joh87] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 154–173. Springer-Verlag, September 1987.

[Kas80] Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, 13:229–256, 1980.

[Kas91a] Uwe Kastens. Attribute grammars as a specification method. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 16–47. Springer-Verlag, 1991.

[Kas91b] Uwe Kastens. Implementation of Visit-Oriented Attribute Evaluators. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 114–139. Springer-Verlag, 1991.

[KS87] Matthijs Kuiper and Doaitse Swierstra. Using attribute grammars to derive efficient functional programs. In *Computing Science in the Netherlands CSN'87*, November 1987.

[KS98] Matthijs Kuiper and João Saraiva. Lrc - A Generator for Incremental Language-Oriented Tools. In Kay Koskimies, editor, *7th International Conference on Compiler Construction*, volume 1383 of *LNCS*, pages 298–301. Springer-Verlag, April 1998.

[KW94] Uwe Kastens and William Waite. Modularity and reusability in attribute grammar. *Acta Informatica*, 31:601–627, June 1994.

[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.

[Mar97] Simon Marlow. *Happy User Guide*. Glasgow University, December 1997.

[Paa95] Jukka Paakki. Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[Pen94] Maarten Pennings. *Generating Incremental Evaluators*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, November 1994.

[PSV92] Maarten Pennings, Doaitse Swierstra, and Harald Vogt. Using cached functions and constructors for incremental attribute evaluation. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *LNCS*, pages 130–144. Springer-Verlag, 1992.

[RT89] T. Reps and T. Teitelbaum. *The Synthesizer Generator*. Springer, 1989.

[SA98] S. Doaitse Swierstra and Pablo Azero. Attribute Grammars in a Functional Style. In *Systems Implementation 2000*, Berlin, 1998. Chapman & Hall.

[Sar] João Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands. (In preparation).

[SKS97] João Saraiva, Matthijs Kuiper, and Doaitse Swierstra. Specializing Trees for Efficient Functional Decoration. In Michael Leuschel, editor, *ILPS97 Workshop on Specialization of Declarative Programs and its Applications*, pages 63–72, October 1997.

[SS99] João Saraiva and Doaitse Swierstra. Data Structure Free Compilation. In Stefan Jähnichen, editor, *8th International Conference on Compiler Construction*, LNCS. Springer-Verlag, March 1999. (to appear).

[VSK89] Harald Vogt, Doaitse Swierstra, and Matthijs Kuiper. Higher order attribute grammars. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24, pages 131–145. ACM, July 1989.