Attribute Grammars as a Specification Method

Uwe Kastens

University of Paderborn

Abstract

Attribute Grammars (AGs) are a formal and practical method for rule based specifications of computations on tree structures. A typical application area is the analysis and translation of formal languages. In this paper we first point out the basic concepts of computations in tree contexts and their dependencies. Then more elaborate methods are presented for systematic development of AG specification.

Longterm experience with AG specification has lead to certain paradigms for attribution. They are presented in terms of typical dependency patterns. Some AG systems provide specific constructs which allow to apply these paradigms in short and comprehensible notations. Realistic applications of AGs are rather large and hence they must be well structured. We present a rather simple but effective module concept for attributions. Finally the use of abstract data types (ADTs) in AGs is introduced. It is shown how sequencing required for ADT operations is systematically specified by attribute dependencies. The examples in this lecture are taken from compiler-specifications formulated for the LIGA system.

1 Introduction

During their more than 20 years old history attribute grammars (AGs) have proven to be a suitable method for specification of computations in tree structures. There is wide range of application areas for AGs with programming language analysis and translation and compiler construction being the most important of them. Systematic techniques for

deriving efficient implementations from AGs have been developed and many generating systems are build around this method. An overview over the state of the art in AGs can be found in [DJL88].

It is clear that complex software systems cannot be developed just on the base of an abstract calculus for algorithms; software engineering methods, paradigms for programming in the large and in the small supported by suitable implementation languages are needed. The same holds for AG specifications of really practical problems, e.g. specifications of a compiler frontend for languages like Pascal or Ada. It is not sufficient just to apply the basic AG concepts.

It is often argued whether AGs are an adequate method for the solution of practical problems. The main arguments presented against the use of AGs in practical specifications are:

- Some AG specifications are as large as or even larger than a manually developed implementation for the same task.
- The concept of locality in AGs requires much redundant information reducing the comprehensibility of the specification.
- Large AGs lack a structure which improves comprehensibility and maintainability.
- The strict functional and declarative character of AGs discourages the use of certain well known efficient implementations, hence yielding less efficient solutions.

The first three arguments are true if only the basic AG concepts are considered, and specifications are written in a notion for exactly that base. But basic AG concepts are the wrong level for that discussion. In the same sense it would be completely impracticable to develop complex software in terms of a Random Access Machine, which is the theoretic model for imperative programming.

AGs define basic concepts for a certain class of specifications. In this paper we show that specification languages for practical use exist which provide suitable notations and structures on top of these concepts. Furthermore they support certain general paradigms for specification development - like programming languages supporting software development paradigms. If such an AG specification language is used in a systematic design

the above arguments will not hold for the result. After an introduction of the use of basic AG concepts in specifications (Sect. 2) we present in Sect. 3 three general attribution paradigms and their notational support in a specification language. A simple and effective modularization concept for AGs is shown in Sect. 5.

The last of the above four arguments is of a different kind. It is correct that certain computations, e.g. for name analysis according to scope rules [KaW90], is more efficiently implemented by a module with a local data structure modified by operations, than by a strictly functional implementation. It is a widespread misunderstanding that the declarative character of AGs prohibits the use of such modules: An AG is a declarative specification of dependencies between computations associated to tree contexts. Two computations are either dependent by definition and use of values, or by pre and post conditions describing the effects of computations. In both cases the dependencies are expressed by attributes. In the latter case they do not propagate a value. In Sect. 4 we show that the use of such state changing implementations of abstract data types (ADTs) can be systematically specified in AGs without loss of its declarative character.

2 Basic Concepts

An AG specifies context dependent computations on tree structures, which are described by the underlying context-free grammar (CFG). Specifications of computations are associated to its productions. Results or effects of computations are described by association to symbols of the CFG. Dependencies between computations are expressed by definitions and uses of attributes. An AG specification does not contain any explicit sequencing of the computations apart from those functional dependencies. If certain formal restrictions hold an evaluator can be derived systematically from an AG specification. Given a particular tree according to the CFG it executes the specified computations in a suitable order. It stores their intermediate results as values of attribute instances, which are implemented by data objects of specified types in suitable locations, e.g. tree components, variables, or stacks.

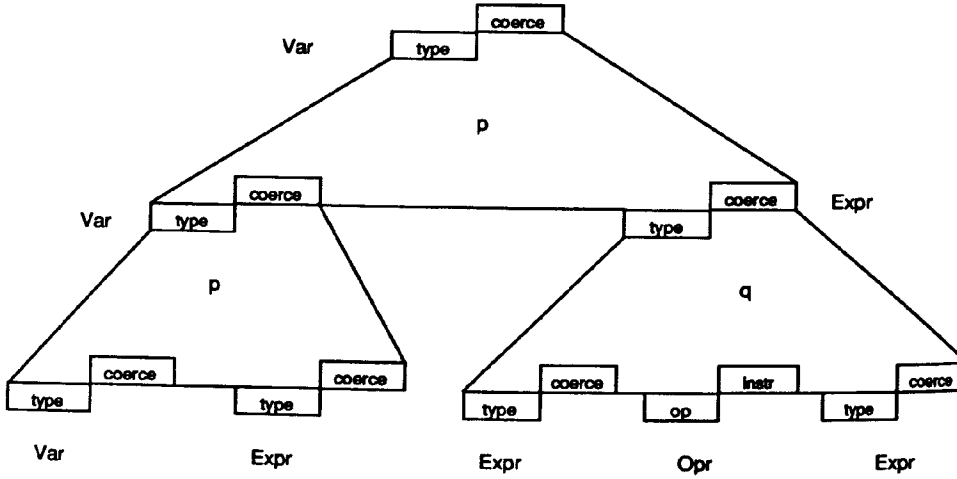
In the following we introduce the basic concepts of AGs from the view of a means for specifications. (For a definition of AGs as a formal calculus the reader is referred to

[Alb91], [DJL88], [Kas80].) We will characterize the specification of dependent computations, introduce the concept of contexts in trees, and describe the role of attributes.

The CFG is the skeleton of the AG. It specifies the trees in which attribute evaluation is executed. Attributes are associated to its symbols and computations are associated to its production contexts. The CFG should be designed such that the attribution can be specified as clear as possible and without unnecessary redundancy. Especially CFG terminals without relevant information and certain chain productions can be omitted from the CFG. Hence in case of compiler specification the CFG should be an **abstract syntax** derived from the **concrete syntax** which is used for parsing and tree construction. In [Kas91] guidance is given for the CFG design with respect to attribution.

The tree nodes are instances of CFG symbols with their attributes. At each nonterminal node a certain production is applied. We say that a node together with its immediate descendants is an **instance of the context of a production** p. (In the following we often omit the term **instance of** where it is clear that we refer to objects of the tree rather than to those of the AG or CFG.) Fig. 2.1 shows a part of a tree with two applications of production p and one of q. The production contexts are indicated by the trapezoid shapes. Each nonterminal node (except the root) belongs to two adjacent contexts; one corresponding to the symbol occurrence on the righthand side of a production (like *Expr* in p), and one given by its derivation (q for this *Expr* node).

For each attribute of a symbol there is an attribute instance at each node for that symbol in the tree. In the design of an AG attributes should be chosen such that each describes a certain **property of a symbol** (and hence of the subtree derived from it). Our example is taken from the type checking task of a compiler. For that purpose two attributes are associated to both *Var* and *Expr*. One describes its type, the second a coercion operation (like contents of an address) to be applied to the construct if necessary. The *Opr* has attributes *op* and *instr* describing the source and the target operator respectively. In an AG specification the association of attributes to symbols is usually stated explicitly together with the attribute types, e.g.



p: Var ::= Var '[' Expr ']'

q: Expr ::= Expr Opr Expr

Fig. 2.1 Production Contexts in a tree

```
NONTERM  Var, Expr:  type:   TypeRep,
                             Instr;
NONTERM  Opr:        op:     Symb,
                     instr:  Instr;
```

Computations of attribute values are context dependent. Specific attribution rules are associated to each production context. In the context *p* of Fig. 2.1 the type of the whole construct (*Var* on the lefthand side of *p*) is computed as the element type of the indexed variable. Hence an attribution rule

Var[1].type = Elem_type_of (Var[2].type)

is associated to context p. It is applied at any instance of p in the tree. Such a computation is specified by a functional expression over attributes and literals. (Its notation and the specification of the functions may vary for different AG specification languages.)

Such a computation may access any attribute of the associated context. Hence the attributes of a nonterminal are accessible from two adjacent contexts. In order to guarantee that each attribute instance is uniquely defined for any pair of adjacent contexts an attribute belongs to one of two classes: **Synthesized** attributes are defined in the "lower context" with their symbol on the lefthand side of the production. They represent properties determined by subtrees derived from the symbol. **Inherited** attributes are defined in the "upper context" with their symbol on the righthand side of the production. They represent properties determined by contexts the subtree is embedded in. In the design of an AG it is absolutely necessary to find out which class the property described by an attribute belongs to. In the graphic of Fig. 2.1 the synthesized attributes (*type, op*) are placed below and the inherited (*coerce, instr*) above the border line of the context. Hence those attributes which must be defined in a specific context lie within its shape.

So the complete attribution of production p in Fig. 2.1 could be specified as

```
RULE p: Var ::= Var '[' Expr ']'
STATIC
   Var[1].type = Elem_type_of (Var[2].type);
   Var[2].coerce = noInstr;
   Expr.coerce = Coercion (Expr.type,
                           Index_type_of (Var[2].type));
END;
```

The attribution of a production p constitutes a **computation pattern** over attribute occurrences in the context p. It is applied at any instance of p in the tree. Each attribute rule also specifies a functional dependency between the attributes used in the expression and that being defined. An attribute evaluator constructed for the AG arranges the computations in an order which is compatible with those dependencies. The order of attribute rules in the specification is completely irrelevant for evaluation. In the above example the three computations may be executed in any order. In general they are

interleaved with computations in adjacent context. The attribution of a production also constitutes a **dependency pattern** over attribute occurrences. Evaluator construction transforms it into an **evaluation pattern** (e.g. a visit-sequence) where the order of computations is fixed.

In the above example each computation yields a value which is stored in an attribute instance and used in other computations of adjacent or of the same context. Their definition-use relation specifies dependencies between the computations. An AG specification may as well contain computations which cause an effect instead of yielding a value. Then attributes are used for only specifying functional dependencies between such computations without carrying a value.

Producing error messages is a simple example for computations which have an effect but do not yield a value. As a basic rule for AG design any computation should be safely executable even if context dependent conditions are violated. Hence no other compilation depends on the check for an error condition. So we formulate such a check as an expression which conditionally produces a message. The attribution of *p* above may be augmented by

```
message_if (NOT (Compatible_types (Expr.type,
                                   Index_type_of (Var[2].type))),
            "wrong index type");
```

This computation depends on the used attributes, but it does not establish a precondition for other computations. Hence no attribute is defined.

A typical example for functional dependencies between computations without results is the specification of output. Assume that we want to produce a postfix notation for expressions like those of production *q* above. We use a function *put* for output of a single symbol. The output sequence is described by two attributes of *Expr* having the following meaning:

pre: the output sequence before this expression is completed

post: the output sequence of this expression is completed

The following attribution specifies the desired sequencing of *put* operations for expressions

```
RULE q : Expr ::= Expr Opr Expr
STATIC
   Expr[2].pre  = Expr[1].pre;
   Expr[3].pre  = Expr[2].post;
   Expr[1].post = put(Opr.op) DEPENDS_ON Expr[3].post;
END;
```

The dependencies specified here establish the preconditions for the two subexpressions. The last computation explicitly DEPENDS_ON the completion of output for the right subtree, and establishes the post condition for the whole expression. (The notation of this example will be drastically simplified by constructs described in Sect. 3.)

The attributes in this example specify functional dependencies, but they do not carry any value. (Their type may be specified VOID.) They contribute to the dependency pattern of the context *q*. Their computation will be eliminated from the evaluation pattern of the evaluator, which contains only the *put* operation at a suitable position. The evaluator does not provide any storage for instances of those attributes. A more general use of such dependency attributes is presented in Sect. 4.

3 Attribution Paradigms

An AG is a declarative specification of computations in recursively defined tree structures. As well as in any other kind of specification or programming there are many different ways to express the same computation. Good AG design should aim at clear and comprehensible specifications without unnecessary redundancy. As a drastic example it is well-known from [Knu68] that any AG could be rewritten such that only synthesized attributes are used. For practical AGs the result would be a rather complicated specification describing properties in an unnatural way.

Experience with AG design especially in compiler specification has led to a set of paradigms and attribution schemes which contribute to clear and comprehensible specifications. Some AG specification languages (like ALADIN for the GAG-System [KHZ82] or LIDO for the LIGA-System [Kas89]) support those paradigms by specific higher level constructs. Their application reduces the size of the AG drastically by elimination of

redundancy. The constructs can be systematically transformed into the basic AG concepts. Hence they improve the specification quality without loss of generality, declarative character, and formal properties of AGs.

A general paradigm of AG design says to consider an attribute as a property of the symbol which it is attached to. Attribution rules specify computations of such properties. All attribute instances are computed exactly once. Their values do not change after being computed (single assignment rule). Hence AG design should be guided primarily by the relationship of properties and computations (expressed by dependencies), rather than starting from an idea of programming computations in a certain sequence. The naming convention for attributes should support that view of properties: Attributes describing the same property of different symbols should have the same name.

The basic AG concepts associate computations to a rather small context given by a single production. On the one hand this locality supports comprehensibility of single computations. On the other hand computations of certain properties often depend on attributes rather far away in the underlying tree. A typical example for long range dependencies is the specification of scope rules, i.e. the relation between definition and application of entities. Those dependencies can of course be broken down into several single attribution rules for adjacent contexts covering the distance between source and target of the information. As a typical result the AG would be scattered with lots of trivial attribution rules like

$$\mathrm{Y.a} = \mathrm{X.a}$$

called **transfer rules** which just propagate identical information. Furthermore attributes have to be associated to symbols in the contexts between source and target for the purpose of information propagation only, so called **transfer attributes**.

In most cases such an attribution belongs to a certain scheme ranging over a larger context. The quality of AG specifications improves drastically if the schemes are expressed directly using higher level specification constructs. The redundant transfer attributes and transfer rules are avoided. They can be generated automatically by the AG system, and they can be object to optimization of the evaluator. In the following sections we present such attribution schemes together with notations used in specification languages ALADIN

and LIDO.

A second class of transfer rules stems from local relations between attributions: Assume that two symbols coincide in a subset of their properties. One symbol occurs on the lefthand side of a production, the other on the righthand side, and the context requires that the corresponding properties are equal. This situation would be specified by a set of transfer rules for those attributes. Again such a local redundancy can be reduced by a higher level specification construct. In the following example both *Var* and *Expr* have the attributes *type* and *coerce*.

```
RULE r: Expr ::= Var
STATIC  TRANSFER;
END;
```

The abbreviation TRANSFER stands for the two transfer rules

```
Expr.type  =  Var.type;
Var.coerce =  Expr.coerce;
```

The direction of the transfer can be deduced from the attribute classes. The attribution may be augmented by further rules for properties which are not identical. The TRANSFER may be restricted to explicitly enumerated attributes, or to some of the symbols on the righthand side of the production.

The use of the TRANSFER construct relies upon application of the above mentioned naming convention for attributes. The above example in fact describes a typical syntactical and semantical chain production. If the distinction between *Expr* and *Var* is not needed in other contexts that production may be eliminated in the design of the abstract syntax.

3.1 Remote Access to an Including Symbol

This attribution scheme is described as follows:

A computation in a context p depends on a property of a symbol X from which p is indirectly derived. In terms of the tree the computation refers to an attribute instance of an X node which is the root of the smallest subtree containing the p instance (but not being the root of the p instance).

A typical example is taken from scope rule specification: In a block structured language the definition of an applied identifier has to be found in the environment of the smallest enclosing block. Hence the identification in the context of an identifier application depends on the environment attribute of the next upward block node in the tree. Fig. 3.1 shows two nested blocks with several identifier contexts enclosed. The arrows represent the dependencies specified by the remote access. The attribution for this example is completely specified by computations for the contexts given in Fig. 3.2.

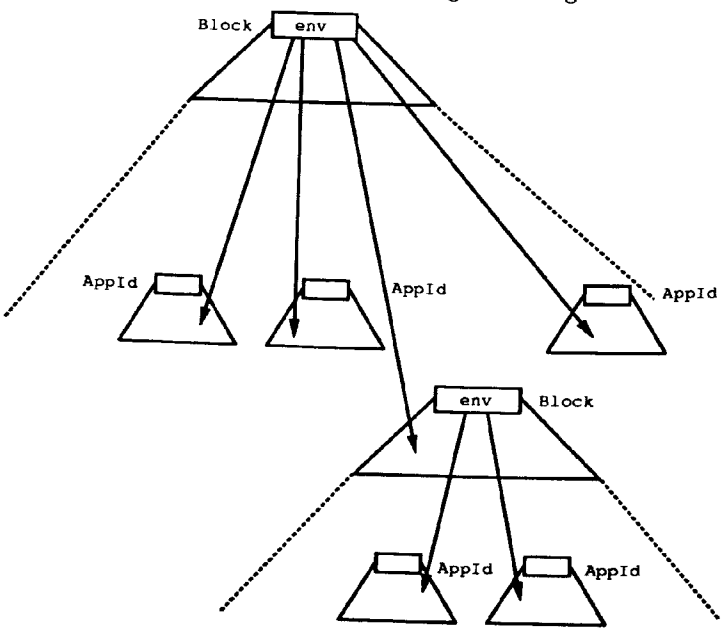


Fig. 3.1 Remote Access to an including symbol

```
RULE p1: Prog ::= Block
STATIC
   Prog.env = NewEnv ();
END;
RULE p2: Block ::= 'begin' Decls Stmts 'end'
STATIC
   Block.env =
    EnterDefs (NewScope (INCLUDING (Block.env, Prog.env)),  Decls.defs);
END;
RULE p3: AppId ::=  Ident
STATIC
   AppId.key = KeyInEnv (INCLUDING (Block.env), Ident.id);
END;
```

Fig. 3.2 INCLUDING for remote access

The attribution of Fig. 3.2 uses the INCLUDING constructs of the specification languages ALADIN and LIDO denoting this kind of remote access. The function *NewEnv* creates an environment for the root context *p1*. The function call *NewScope (e)* creates a new environment embedded within the environment *e*. In *p2* the argument is stated by the INCLUDING construct. It yields the environment of the surrounding *Block* or that of *Prog*, depending on which is the smallest enclosing structure. *EnterDefs* enters the definitions of that block into the environment. In *p3 KeyInEnv* searches the definition in the environment of the smallest enclosing *Block*.

In general three syntactically different situations can be distinguished. Let p be the context with the remote access, and let X be its lefthand side symbol.

1. The remote context is unique, i.e. in any tree the context p is embedded in a subtree rooted by the source symbol of the remote access, e.g. *INCLUDING (Block.env)* of *p3* in Fig. 3.2.

2. Alternative symbols are specified for the remote source, e.g.
   INCLUDING (Block.env, Module.env, WithStmt.env).
   Here the remote source is the smallest p enclosing subtree rooted either with *Block*, *Module*, or *WithStmt*. (In this case attribution may be simplified if in the abstract syntax a single symbol can be introduced for each of the alternatives, leading to case (1) then.)

3. The remote source symbol is X, the same as the lefthand side of p i.e. the remote access follows the recursive derivation of X referring to the next upward instance of X in the tree. The recursion is terminated by an alternative source, which often is the root symbol. This case applies to the *INCLUDING (Block.env, Prog.env)* in p_2 of Fig. 3.2.

All transfer attributes and transfer rules for intermediate contexts are avoided by the described constructs. They can be deduced automatically, e.g. for dependency analysis. Furthermore there are implementation techniques which may omit them in the evaluator. If the remote access is used to specify a dependency only, rather than a value access, the construct and its transfer expansion also vanishes from the evaluator.

3.2 Remote Access to Subtree Components

This attribution scheme is described as follows:

> A computation in a context p depends on a property of all instances of a symbol X in the subtree rooted by the p instance. If the lefthand side Y of p is recursive, then no X instances in inner subtrees rooted by Y contribute to the computation in the outer p instance.

A typical example for this scheme is again taken from scope rule specification: In a block structured language each definition contributes to the environment of the enclosing block. Hence the computation of an environment attribute depends on all definitions which are constituents of the block but not of inner blocks. Fig. 3.3 shows two nested blocks with several definitions. The arrows represent the dependencies specified by the remote access. Using this scheme the attribution of Fig. 3.2 is refined in Fig. 3.4

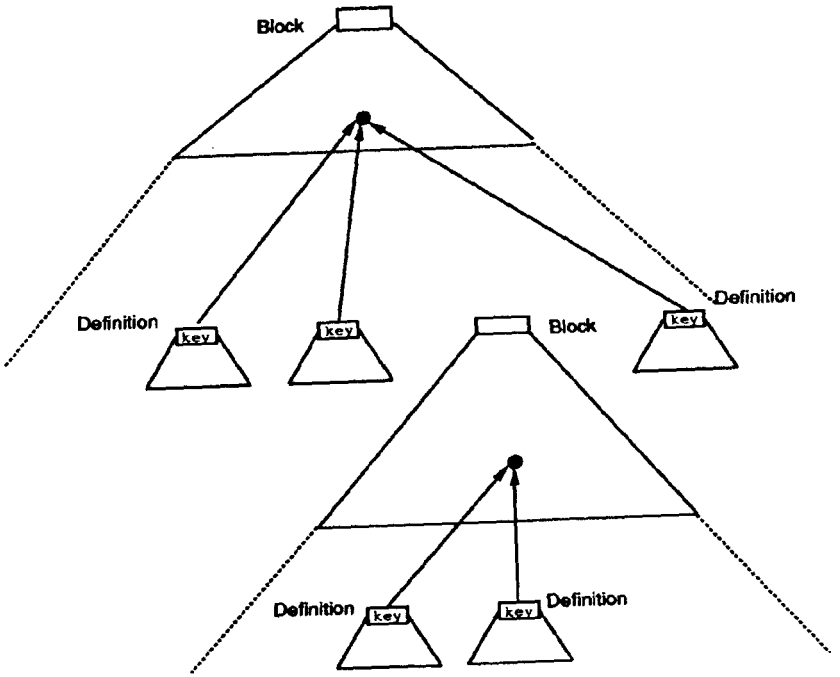


Fig. 3.3 Remote access to subtree components

```
RULE p2: Block ::= 'begin' Decls Stmts 'end'
STATIC
   Block.env =
      EnterDefs (NewScope (INCLUDING (Block.env, Prog.env)),
                 CONSTITUENTS Definition.def
                    WITH DefList, DefAppend, DefSingle, DefEmpty);
END;
RULE p4: Definition ::= Type Ident ';'
STATIC
   Definition.def = MakeDef (Ident.id, SomeProperties);
END;
```

Fig. 3.4 CONSTITUENTS for remote access

The attribution of Fig. 3.4 uses the CONSTITUENTS construct of the specification languages ALADIN and LIDO denoting this kind of remote access. Instead of passing all definitions up to *Decls.def* as in Fig. 3.2, they are collected by the remote access *CONSTITUENTS Definition.def* from the *Definitions* in the subtree. According to the description of the scheme definitions of inner blocks do not contribute to the CONSTITUENTS of this context instance. They may contribute to such a construct associated to their context.

All transfer attributes and transfer rules for intermediate contexts are avoided by the described construct. They can be deduced automatically. We assume here that lists of type *DefList* are formed by user defined functions *DefAppend, DefSingle, DefEmpty*. *DefSingle* makes a list from a single element *Definition.def*. *DefAppend* concatenates two lists; it is applied left associative to the list values of subtrees. *DefEmpty* makes an empty list; it is called in contexts $A ::= u$ where u does not derive to *Definition*, if there is a context $A ::= w$ where w does derive to *Definition*. The names of the type and of these functions are associated to the CONSTITUENTS construct. The resulting list contains the *Definition.def* values in an order which corresponds to a postorder of the *Definition* nodes in the subtree.

Completely different computations can be specified by other choices of attribute types and composition functions: For example a construct like

CONSTITUENTS Stmt.instr WITH InstrList, Append, Single, Empty

yields an instruction list for the statements of the subtree, whereas

CONSTITUENTS Stmt.costs WITH int, Add, Identity, Zero

computes the sum of all accessed statement costs.

If the CONSTITUENTS construct does not describe a value dependency, no computation is specified, and so the WITH part with type and function names is omitted. In Sect. 4 the above example is modified in that way.

3.3 Chaining

Chaining is a frequently applied attribution scheme described as follows:

> Some symbols have a certain property *a*. The instances of *a* within a subtree depend on each other in a depth-first, left-to-right order of the symbol instances.

There are two typical variants of the scheme. An example for the first is given by computing a storage map for the variables of a *Block*. The address of each variable is a property which depends on the left-to-right order of the *Definitions*. Each *Definition* symbol has the property representing the next free address at the symbol instance. Its computation depends on the result of the mapping at the *Definition* instance left to it, or on an initialization in the block context for the left most *Definition*. Fig. 3.5 (a) shows the idea of such dependencies. The arrows represent computations of that property associated to the definition context, except the upward arrow which represents the use of the result in the block context.

In order to make the chaining scheme more obvious in Fig. 3.5 (b) the expansion of the example in terms of basic attribution concepts is given. The property is in fact described by a pair of attributes: The inherited attribute *Definition.pre* represents the next free address for this symbol instance, the synthesized attribute *Definition.post* represents the next free address after allocation in this context. Such attribute pairs are introduced at all intermediate nodes for transfer as indicated. The example shows that an application of the scheme is completely described by the following specifications:

a) introduction of the chained property, here the next free address,

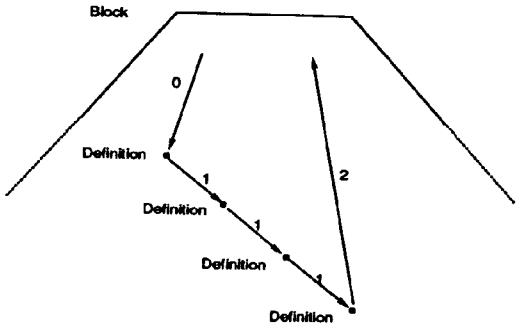
b) indication of the root context for the subtree, here the *Block* context,

c) initialization of the chain in the root context, arrow (0) here,

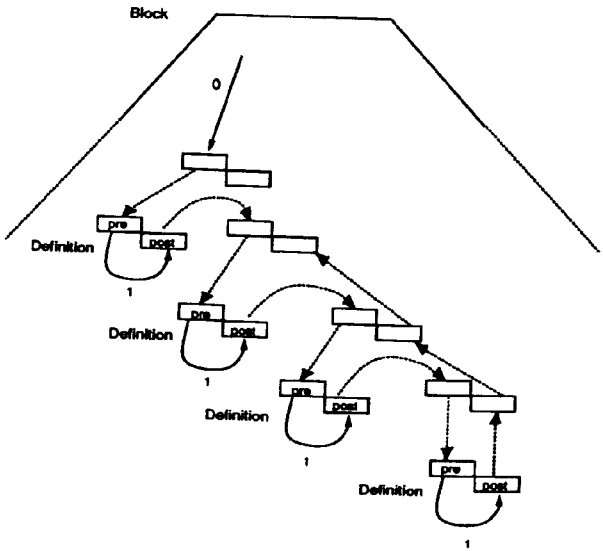
d) computation of the chained property at symbols in the subtree, here arrows (1) in the *Definition* context.

The mapping to basic concepts with attribute pairs and transfer rules wherever required (Fig. 3.5 (b)) can be deduced automatically.

In Fig. 3.6 an AG notation in LIDO is given for this example. Here the chained property describes a value dependency. In the attribution the name of the CHAIN is used as attribute names wherever the CHAIN is referenced. The occurrence of a CHAIN



(a) Definition chain dependencies



(b) Definition chain expanded

Fig. 3.5 Chaining

```
CHAIN  addr: int;                                          (a)
RULE  Block ::= 'begin' Decls stmts 'end'
STATIC
   CHAINSTART  Decls.addr = BlockBaseAddr;                 (b,c)
END;
RULE  Definition ::= Type Ident
STATIC
    Definition.addr =                                      (d)
       Increment (Definition.addr, Type.size);
END;
```

Fig. 3.6 Chain attribution in LIDO

reference either on the lefthand side of an attribute rule (defining) or on the righthand side (applied), and its symbol being on the lefthand side or righthand side of the production, distinguish the mapping to the *pre* or to the *post* attribute. Hence the attribute rule (d) is not cyclic but establishes a dependency from *Definition.pre_addr* to *Definition.post_addr*, cf. Fig. 3.5 (b).

It should be noted that there is no need to specify explicitly which parts of the subtree are reached by the CHAIN. The CHAIN would lead through *Stmts*, too, in the above example only if *Stmts* contain references to *defs*. If the root context (*Block*) is recursive each instance has its own separated instance of the CHAIN. In the above example the symbols which conceptually have the chained property (*Definition*) are not recursive. Its instances lie on a cut line of the subtree.

The second typical variant of chaining is a generalization, including recursion and CHAIN accesses in arbitrary contexts of the subtree. It is demonstrated in Fig. 3.7 by an attribution which produces output of expressions in post order. Attribute rule (3) specifies a computation inserted in the upward chain dependency of the recursive *Expr* context, producing the output of the operator after the output of the right operand is produced. Computation (2) depends on the "end of the CHAIN", producing the final ":=" operator. The example could be modified to output in inorder, by simply replacing

```
CHAIN  out : VOID;
RULE  Stmt ::= Ident ':=' Expr
STATIC
   CHAINSTART  Expr.out = put(Ident.id);                    (1)
   put(":=") DEPENDS_ON Expr.out;                           (2)
END;
RULE  Expr ::= Expr Opr Expr
STATIC
   Expr[1].out = put(Opr.op) DEPENDS_ON Expr[3].out;        (3)
END;
RULE  Expr ::= Ident
STATIC
   Expr.out = put(Ident.id) DEPENDS_ON Expr.out;            (4)
END;
```

Fig. 3.7 Output of postfix expressions by chaining

rule (3) by

```
Expr[3].out = put (Opr.op) DEPENDS_ON Expr[2].out;
```

and replacing rule (1) and (2) by

```
CHAINSTART Expr.out = ORDER (put(Ident.id), put(":="));
```

where *ORDER (a, b)* specifies execution of first *a* and then *b*. Other typical examples for applications of the chaining scheme are

- consecutive numbering of block instances or other language constructs,
- pre and post conditions of statements with respect to some property (e.g. in data flow analysis),
- any computation of sequential lists in left to right order of tree nodes.

4 Abstract Data Types

An abstract data type (ADT) defines a type of concern by a set of functions which construct objects of that type or access values from such objects. ADTs are implemented by program modules, thus providing an effective method for structuring software. An ADT may be implemented strictly functional such that its operations are free of side effects. The constructor functions of an ADT can also be implemented such that they modify a data structure local to the module. Then their effect causes a transition of the ADT state. Such implementations are well suited and efficient for many ADT implementations, like "dictionaries" associating properties to object keys, or representation of environments in a compiler.

Using ADTs in AG specifications improve AG design and implementation in several aspects: The AG specification concentrates on the central concept of AGs - specifying functional dependencies between computations which use the ADT operations. There is a clear interface between the AG computations and the ADT implementations. Function calls pass arguments by value and may return values. If an ADT implementation imposes restrictions on the sequence of calls of state transition functions, they are specified by dependencies in the AG (without propagating a value). The implementation of ADT operations and their data structures is separated from the AG itself. As a consequence ADT implementation can take any advantage from the chosen implementation language, like modular decomposition, separate compilation, and techniques for implementation of complex data structures and functions. By those means efficient implementations can be achieved without loss of the declarative character of the AG specification.

ADTs in AG specifications were first suggested by Waite in [Wai86] where externally defined types and functions were introduced in the specification language ALADIN. For the LIGA system and its specification language LIDO this separation is a central concept: All types and functions which are used in the AG are implemented outside of the AG.

In the following we demonstrate several basic techniques for using ADTs in AGs by simple but typical examples. We start from a strictly functional ADT implementation, and then show different application schemes for state transition ADT implementations.

An example for a **strictly function ADT module** is a module for linear lists. It

provides a data type *LinList* and functions *Cons, Head, Tail,* and *Empty* with the usual meaning. Since those functions do not have any side effects, and each value is ever accessible after its construction, their use in AG computations is either expressed by nested function calls in single attributes rules like

$$X.l = Cons\ (Y.a, Tail\ (Z.l));$$

or by the value dependencies between definition and use of attributes. Those attributes have the type *LinList.* The attribute evaluator provides storage for them, whatever implementation of that type is chosen in the ADT module, e.g. a pointer or a pair of pointers.

In general a **state transition ADT module** implements a set of functions which modify or access a data structure local to the module. Hence some constructor functions perform an irreversible transition of the module state. The access functions yield values of the data structure which depend on the actual state. Hence the sequence of function calls has to obey restrictions in order to yield the desired results: certain module states are pre and post conditions of AG computations. Attribute dependencies are a natural way to specify such restrictions as precise as required.

We again start from a trivial example which imposes no such ordering restrictions. Let us assume there is a module which collects error messages during attribute evaluation. It implements a single operation

message_if (condition, message_text).

After completion of attribute evaluation finalization causes insertion of the messages into the source text. Hence there is no restriction on the order of the calls of the message function, and it does not return a value. Such a function call is specified by a computation in appropriate production contexts without defining an attribute, e.g.

message_if (is_multiple (Def.key), "multiple definition");

The AG system is free in its choice of the evaluation order for these calls. This message

ADT can be considered as a state transition implementation: Each call of the message function modifies its internal data structure, and hence alters the state. But the specification does not require any restriction of state sequence. The ADT states are ignored in the AG.

In Sect. 3 we presented an example for translation of expressions into postorder form. It can be understood as an application of a very simple state transition ADT module. The *put* operation is its only state transition function. There are very tight sequencing restrictions to achieve postorder representation: The precondition for output of an operator is the completion of the output of its operands and any expression left to it. Its postcondition completes the output of the expression for that operator. The precondition for output of a leaf operand is the completion of all expressions left to it. These pre- and postconditions model states of the ADT with respect to the tree structure. They lead directly to the CHAIN specification in Sect. 3, and to the meaning of the pre- and post-attributes introduced by it.

The next example shows a simple state transition ADT where only two of its arbitrary many states are relevant for specification. The ADT module implements a simple counter which is used for counting the instances of a certain language construct, e.g. assignments. The ADT has the operations *Init*, *Incr* and *Print* with obvious meanings. *Init* and *Incr* are state transition functions, *Print* is an access function. The desired calling sequence can be described by the regular expression

Init Incr* Print

The *Incr* calls may occur in any order between the calls of *Init* and *Print*. Hence the precondition of each *Incr* call is "*Init* has been called". The precondition of *Print* is "all *Incr* calls have occurred". This view leads immediately to the attribution of Fig. 4.1. The two states - after *Incr* and before *Print* - are represented by the attribute *Prog.init* and the CONSTITUENTS constructs in the root context.

A more elaborate specification of state transitions is required if the counter should be reused for counting assignments of each (possibly nested) procedure in the program separately. The calling sequence then has to be

(Init Incr* Print)*

```
NONTERM Prog : init : VOID;
RULE Prog ::= ...
STATIC
   Prog.init = Init();
   Print() DEPENDS_ON CONSTITUENTS Assign.incr;
END;
NONTERM Assign: incr : VOID;
RULE Assign ::= ...
STATIC
   Assign.incr = Incr() DEPENDS_ON INCLUDING (Prog.init);
END;
```

Fig. 4.1 Counting language constructs

where the outer iteration corresponds to the procedure instances. Now *Print* establishes the precondition for the next *Init*. The attribution in Fig. 4.2 is derived from that in Fig. 4.1. The counting sequences for each procedure are chained. The CHAIN attributes *cnt* represent the states between *Print* and *Init*, meaning completion of one counting sequence. (It should be noted that this attribution is slightly over specified, since the procedures are considered in postorder.)

In our last example we present an application of a state transition having both constructor and access functions. Results of the access function depend on the state. The example also shows abstractions of the states in different granularities.

The example describes the definition of objects and association of properties to them, i.e. a typical task of a definition module in a compiler, or some kind of dictionary in other applications. It may be implemented by property lists for example. Such a module is provided in the compiler construction environment Eli [WHK88]. The ADT operations are

```
CHAIN  cnt : VOID;
RULE  Prog ::= Block
STATIC
   CHAINSTART Block.cnt = NoOp ();
END;
NONTERM  Procedure: init: VOID;
RULE  Procedure ::= Block
STATIC
   Procedure.init = Init() DEPENDS_ON Block.cnt;
   Procedure.cnt = Print() DEPENDS_ON CONSTITUENTS Assign.incr;
END;
NONTERM Assign: incr: VOID;
RULE  Assign ::= ...
STATIC
   Assign.incr = Incr() DEPENDS_ON INCLUDING (Procedure.init);
END;
```

Fig. 4.2 Counting language constructs in nested structures

SetProp (key, newval, chgval)

associates a value for the property *Prop* to the object identified by *key*. If that property has already a value for that key *chgval* is associated, otherwise *newval*.

GetProp (key, default)

returns the value of the property *Prop* for the object identified by *key*, if it is set previously, otherwise the *default* value.

The *SetProp* operation modifies (the state of) the module's data structure. The application usually requires that if there is a *SetProp* call for a particular *key* and *Prop* pair it should precede all *GetProp* calls for the same pair.

The attribution in Fig. 4.3 checks definitions and applications of identifiers. The ADT

```
NONTERM Prog: def, use : VOID;
RULE  Prog ::= ...
STATIC
   Prog.def = CONSTITUENTS Definition.def;
   Prog.use = CONSTITUENTS Application.use;
END;

NONTERM Definition: def: VOID;
RULE Definition ::= 'var' Ident ';'
STATIC
   Definition.def = SetIsDef (Ident.id, defined, multiple);
   message_if (EQ (GetIsDef (Ident.id, undefined), multiple),
                  "multiply defined identifier")
              DEPENDS_ON INCLUDING (Prog.def);
   message_if (EQ (GetIsUsed (Ident.id, unused), unused),
                  "unused definition")
              DEPENDS_ON INCLUDING (Prog.use);
END;

NONTERM Application : use : VOID
RULE Application ::= Ident
STATIC
   Application.use = SetIsUsed (Ident.id, used, used);
   message_if (EQ (GetIsDef (Ident.id, undefined), undefined),
                  "undefined identifier")
              DEPENDS_ON INCLUDING (Prog.def);
END;
```

Fig. 4.3 ADT for property association

associates two properties *IsDef* and *IsUsed* to identifier encodings (attribute *Ident.id*), which play the role of the object key here. The properties may have the values

IsDef: ***undefined, defined, multiple***

IsUsed: ***unused, used***

For ease of presentation we assume trivial scope rules here: Each definition is valid in the whole program. (Generalization to nested scopes is mentioned below.)

In the *Definition* context the property *IsDef* is set to *defined*, or modified to *multiple* if it was already set. The property *IsUsed* is set to *used* in the application context. The corresponding attributes describe the state information "the property is set for that symbol instance". Three conditions are checked in this attribution: An identifier is marked to be multiply defined if there is another definition for that identifier. The precondition for the check is the fact that the *SetIsDef* operations are executed for all *Definitions* of that identifier. The dependency on *Prog.def* describes a less specific state information: *Prog.def* represents a state where all *SetIsDef* operations are executed. It should be remarked that the specification does not require any order of the execution of *SetIsDef* and *SetIsUsed* operations, and that each one of multiple definitions is marked by a message. The same scheme is applied for the other two messages indicating unused and undefined identifiers.

The attribution of Fig. 4.3 specifies that the two properties *IsDef* and *IsUsed* are independent of each other. It is important for clarity and maintainability of the specification not to introduce unnecessary dependencies, which would overspecify the problem and reduce the freedom of evaluation order. In our example the same results could be achieved if in the *Prog* context only one less specific state information is used for the precondition of all three checks, e.g.

Prog.set = CONSTITUENTS (Definition.def, Application.use);

The attribution would be slightly simpler, but the evaluation order is more restricted: All *SetIsDef* and *SetIsUsed* operations must occur before the first *GetIsDef* or *GetIsUsed* operation.

On the other hand there are situations where the state information must be specified as specific as in our example: Assume that in some context one property, e.g. the type of

```
RULE Prog ::= ...
STATIC
   Prog.type_def = CONSTITUENTS TypeDefinition.type;
   Prog.type =     CONSTITUENTS Definition.type;
END;
RULE Definition ::= 'var' TypeIdent Ident
STATIC
   Definition.type =
      SetType (Ident.id,
               GetTypeDef (TypeIdent.id, error_type), error_type)
      DEPENDS_ON INCLUDING (Prog.type_def);
END;
```

Fig. 4.4 Dependent properties

an object, is set by accessing another property, e.g. a type definition, as in Fig. 4.4. This functional dependency would automatically lead to an evaluation order, where first the *TypeDef* properties are set and then the *Type* properties. If the state information would merge the setting of all properties into one state attribute *Prog.set*, as mentioned above, this attribution would be cyclic.

This example can be extended systematically to the specification of block structured scope rules: In that case each definition has to be identified by a unique key, instead of the identifier code. Each identifier in a particular block context is mapped to the key of its definition according to the scope rules of the language. That mapping can be specified using an ADT module which implements the concept of environments for nested scopes. Such an ADT is described in [KaW90]. An implementation belongs to the set of standard modules in Eli [WHK88]. The association of properties to definition keys is then specified as discussed above.

5 Modularization of Attribute Grammars

Complete AG specifications of compilers for complex languages can be very large: e.g. about 2000 lines for the Pascal frontend specified in [KHZ82], or more than 20000 lines for Ada in [UDP82]. Both are written in ALADIN, and they contain the complete specifications of all functions used in the AG, which contribute to about half of the size. Consequent application of the ADT concepts of the previous section would separate the two parts and reduce the size of both. The function part can be well structured into sufficiently small modules for ADTs of different tasks. In this section we concentrate on the modularization of the AG part, which would still be rather large in the above cases.

The basic AG concepts do not provide a means for modularization. The central part of an AG is a sequence of productions, each with its attribution associated. That principle of locality supports comprehensibility of attribute rules in their syntactic context. But it does not allow for any global structure of the AG. One could try to subdivide the AG by syntactic criteria, e.g. parts for attribution of the productions for declarations, statements, and expressions. In general such a structure decreases comprehensibility, because related properties are described in several parts, but should be understood together. For example attribution for type rules occurs in all three of the above parts.

The structuring problem mainly results from the fact that all computations of different semantic aspects for one context are comprised in a single attribution of a production. However each of them is more closely related to attributions of the same property in other contexts than to other attribute rules of the same context. Fig. 5.1 shows part of an expression syntax with attribution for typing in production contexts, p, q, and r, for scope rules in r and for postorder output in p and r.

This observation leads to a very simple and effective modularization concept: A module of the AG contains the attribution of one semantic aspect. It is composed by some production contexts with their attribute rules related to that aspect. The above example would contribute to three attribution modules for scope rules, for typing, and for output. Such a decomposition can be achieved by a simple notational extension: There may be several RULE constructs which associate attribute rules to the same production. Then the modules can be written on one file each. Composing the modules and mapping the

```
NONTERM Expr:          type: Type;
NONTERM Opr:           op: Symb,
                       instr: Instr;
NONTERM AppIdent:      key: Key,
                       type: Type;
TERM Ident:            id: Symb

CHAIN out: VOID;

RULE p: Expr ::= Expr Opr Expr
STATIC
   Opr.instr =    OprIdentif (Opr.op, Expr[2].type, Expr[3].type);
   Expr[1].type = ResultType (Opr.instr);
   Expr[1].out =  put (Opr.instr) DEPENDS_ON Expr[3].out;
END;

RULE q: Expr ::= AppIdent
STATIC  TRANSFER  type;
END;

RULE r: AppIdent ::= Ident
STATIC
   AppIdent.key =  KeyInEnv (INCLUDING (Block.env), Ident.id);
   message_if (EQ (AppIdent.key, NoKey), "undefined identifier");
   AppIdent.type = Get (AppIdent.key, has_type, err_type)
                   DEPENDS_ON  INCLUDING (Block.types_set);
   AppIdent.out =  Put (Ident.id)  DEPENDS_ON  AppIdent.out;
END;
```

Fig. 5.1 Attribution for scope rules, typing and output

attribute rules to their production contexts is an easy task of the frontend of an analyzer for the AG specification language. The same principle can be applied for the association of attributes to symbols: Attributes may be associated to a symbol in several NONTERM or TERM constructs. These notational facilities have proven to be an effective support for modularization in the LIDO specification language.

```
NONTERM  Expr, AppIdent:    type: Type;
NONTERM  Opr:               op:   Symb;
TERM     Ident:             id:   Symb;

RULE p : Expr ::= Expr Opr Expr
STATIC
   Opr.instr = OprIdentif (Opr.op, Expr[2].type, Expr[3].type);
   Expr[1].type = ResultType (Opr.instr);
END;

RULE q : Expr ::= AppIdent
STATIC  TRANSFER  type;
END;

RULE r : AppIdent ::= Ident
STATIC
   AppIdent.type =
      Get (AppIdent.key, has_type, err_type)
      DEPENDS_ON  INCLUDING (Block.types_set);
END;
```

Fig. 5.2 a Part of an attribution module for typing

```
NONTERM  AppIdent:                    key: Key

RULE r : AppIdent ::= Ident
STATIC
   AppIdent.key = KeyInEnv (INCLUDING (Block.env), Ident.id);
   message_if (EQ (AppIdent.key), NoKey), "undefined identifier");
END;
```

Fig. 5.2 b Part of an attribution module for scope rules

```
CHAIN out:  VOID;

RULE p : Expr ::= Expr Opr Expr
STATIC
   Expr[1].out = put (Opr.instr) DEPENDS_ON Expr[3].out;
END;

RULE r : AppIdent ::= Ident
STATIC
   AppIdent.out = Put (Ident.id)  DEPENDS_ON  AppIdent.out;
END;
```

Fig. 5.2 c Part of an attribution module for output

In Fig. 5.2 it is shown how the example of Fig. 5.1 can be decomposed into modules. Each module contains only attributions for one semantic aspect, here typing, scope rule, and output. This example shows only few contexts of a complete specification. Each module would be completed by corresponding attributions of further contexts which contribute to its semantic aspect, e.g. the scope module by attributions of the *Block* and the *Prog* context. Contexts which do not contribute to that aspect are not mentioned in the module, e.g. the *Expr* context *p* in the scope module. The same holds for attribute association to symbols. If additionally the technique of remote attribute access is applied each module contains only those specifications which are really relevant for a certain specification aspect. The modules can thus be designed rather compact and comprehensible.

6 References

[Alb91] Alblas, H., *Introduction to Attribute Grammars*, Proceedings of the International Summer School on Attribute Grammars, Application and Systems, Prague (1991).

[DJL88] Deransart, P., Jourdan, M. and Lorho, B., *Attribute Grammars*, Lecture Notes in Computer Science, vol. 323, Springer Verlag, Berlin, 1988.

[Kas89] Kastens, U., *LIGA: A Language Independent Generator for Attribute Evaluators*, Universität-GH Paderborn, Bericht der Reihe Informatik Nr. 63, 1989.

[Kas80] Kastens, U., *Ordered Attributed Grammars*, Acta Informatica 13 (1980), 229–256.

[Kas91] Kastens, U., *An Attribute Grammar System in a Compiler Construction Environment*, Proceedings of the International Summer School on Attribute Grammars, Application and Systems, Prague (1991).

[KHZ82] Kastens, U., Hutt, B. and Zimmermann, E., *GAG: A Practical Compiler Generator*, Lecture Notes in Computer Science, vol. 141, Springer Verlag, Heidelberg, 1982.

[KaW90] Kastens, U. and Waite, W. M., *An Abstract Data Type for Name Analysis*, accepted for publication in Acta Informatica, 1990.

[Knu68] Knuth, D. E., *Semantics of Context-Free Languages*, Mathematical Systems Theory 2 (June 1968), 127–146.

[UDP82] Uhl, J., Drossopoulos, S., Persch, G., Goos, G., Daussmann, M., Winterstein, G. and Kirchgässner, W., *An Attributed Grammar for the Semantic Analysis of ADA*, Lecture Notes in Computer Science, vol. 139, Springer-Verlag, New York-Heidelberg-Berlin, 1982.

[Wai86] Waite, W. M., *Generator for Attribute Grammars - Abstract Data Type*, Gesellschaft für Mathematik und Datenverarbeitung, Arbeitspapier 219, Karlsruhe, BRD, September 1986.

[WHK88] Waite, W. M., Heuring, V. P. and Kastens, U., *Configuration Control in Compiler Construction*, in International Workshop on Software Version and Configuration Control '88, Teubner, Stuttgart, 1988.