Component-based Programming for Attribute Grammars

João Saraiva
Departamento de Informática
Universidade do Minho
Campus de Gualtar, 4710-057 Braga, Portugal
jas@di.uminho.pt

Pablo Azero
Departamento de Informática y Sistemas
Universidad Mayor de San Simón
Cochabamba, Bolivia
pablo@memi.umss.edu.bo

Abstract

This paper presents techniques for a component-based style of programming in the context of attribute grammars (AG). Attribute grammar components are "*plugged in*" into larger attribute grammar systems through higher-order attribute grammars (HAG). Higher-order attributes are used as (intermediate) "gluing" data structures.

This paper also presents two attribute grammar components that can be re-used across different language-based tool specifications: a pretty-printing and a graphical user interface AG component. Both components are reused in the definition of a simple language processor. The whole processor is specified within a single paradigm and a single notation: (higher-order) attribute grammars. Indeed, the main tasks of the processor, namely: parsing/unparsing, static semantics, semantic functions and graphical user interface, are specified with attributes and attribute equations. The techniques presented in this paper are implemented in LRC: a purely functional, higher-order attribute grammar-based system that generates language-based tools.

1 Introduction

Recent developments in programming languages are changing the way we construct programs. Programs are now a collection of generic, reusable, off-the-shelf program components that are "*plugged in*" to form larger and powerful programs. In such an architecture, intermediate gluing data structures are used to convey information between different program components: a component constructs (produces) an intermediate data structure which is used (consumed) by other component.

In the context of the design and implementation of language-based tools, attribute grammars provide powerful properties to improve the productivity of their users, namely, the static scheduling of computations. Indeed, an attribute grammar writer is neither concerned with breaking up her/his algorithm into different traversal functions, nor is she/he concerned in conveying information between traversal functions (*i.e.*, how to pass

intermediate values computed in one traversal function and used in following ones). A second important property is that circularities are statically detected. Thus, the existence of cycles, and, as a result, the non-termination of the algorithms, is detected statically. That is to say that for (ordered) attribute grammars the termination of the programs for all possible inputs is statically guaranteed. A third characteristic is that attribute grammars are declarative. Furthermore, they are executable: efficient declarative (and non-declarative) implementations (called attribute evaluators) are automatically derived by using well-known AG techniques. Finally, incremental implementations of the specified tools can be automatically generated from an attribute grammar.

Despite these advantages attribute grammars are not of general use as a language-based tool specification formalism. In our opinion, this is due to three main reasons: firstly, there is no efficient, clear and elegant support for a component-based style of programming within the attribute grammar formalism. Although an efficient form of modularity can be achieved in AGs when each semantic domain is encapsulated in a single AG component [GG84, LJPR93, KW94, CDPR98, SS99b, dMBS00], the fact is that there is no efficient support within the AG formalism for an easy reuse of such components. That is, how can a grammar writer "*plug in*" an AG component into her/his specification? How are those AG components *glued* together? How is information passed between different AG components? How can the separate analysis and compilation of components be achieved? Obviously we wish to provide answers to these questions within the attribute grammar formalism itself. Secondly, there is a lack of good generic, reusable attribute grammar components that can be easily "*plugged in*" into the specifications of language-based tools. Components that are themselves written in the AG formalism. The third reason is that there are too many notational issues involved in the writing of large attribute grammars. Indeed, most attribute grammar-based systems provide special notation for: parsing, unparsing, the attribute equations, a declarative language for defining the semantic functions, notation for defining the interface, etc.

In this paper, we show how a style of component-based programming can be achieved within the attribute grammar formalism. Our techniques rely on higher-order attribute grammars [VSK89]: an extension to the classical AG formalism where attributes, the so-called higher-order attributes, are tree-valued attributes. We may associate, once again, attributes with such higher-order trees. Attribute grammar components are efficiently and easily "plugged-in" into an AG specification via higher-order attributes: one AG component defines a higher-order attribute which is decorated according to the attribute equations defined by other AG component for that attribute/tree. The separate analysis and compilation of AG components can be achieved by using the techniques we have introduced in [SS99b]: The HAG component (that reuses other AG component as an higher-order attribute) *knows* the pattern of attribute propagation of the reused component. The pattern of attribute propagation of the reused component is given by the attribute *partitions* [Kas80] infered by the Kastens' ordered algorithm [Kas80] when analysing such attribute grammar independently.

Furthermore, we present two generic, reusable, off-the-shelf attribute grammar components: a pretty-printing and an advanced graphical user interface (GUI) AG component.

A key feature of our approach is that a complete language-based tool can be specified within a single formalism: the higher-order attribute grammar formalism. No specific formalism/notation has to be used to define, for example: the "gluing" of an AG component, the mapping between concrete and abstract syntaxes, the parsing/unparsing, the static semantics, the semantic functions, the graphical user interface. Everything is defined through (higher-order) attributes and attribute equations. By expressing the complete tool specification within the AG formalism, we inherit all the nice properties of attribute grammars. For example, by expressing the interface of the tools via the AG GUI component, we can derive incremental implementations of such tools where the GUIs are incrementally computed. Our techniques are implemented in the LRC system: an AG-based system that generates incremental language-based tools [KS98, SSK00].

This paper is organized as follows: Section 2 presents higher-order attribute grammars, its notation and provides a simple example that will be used throughout the paper. Section 3 introduces AG component-based programming and presents two generic AG components: a pretty-printing component (Section 3.1) and graphical user interface component (Section 3.2). Section 4 discusses related work and Section 5 contains the conclusions.

2 Higher-Order Attribute Grammars

The techniques presented in this paper are based on the *higher-order attribute grammar* formalism [VSK89]. Higher-Order Attribute Grammars are an important extension to the attribute grammar formalism. Conventional attribute grammars are augmented with *higher-order attributes*, the so-called *attributable attributes*. Higher-order attributes are attributes whose value is a tree. We may associate, once again, attributes with such a tree. Attributes of these so-called *higher-order trees*, may be higher-order attributes again. Higher-order attribute grammars have four main characteristics:

- First, when a computation can not be easily expressed in terms of the inductive structure of the underlying tree, a better suited structure can be computed before. Consider, for example, a language where the abstract grammar does not match the concrete one. Consider also that the semantic rules of such a language are easily expressed over the abstract grammar rather than over the concrete one. The mapping between both grammars can be specified within the higher-order attribute grammar formalism: the attribute equations of the concrete grammar define a higher-order attribute representing the abstract grammar. As a result, the decoration of a concrete syntax tree constructs a higher-order tree: the abstract syntax tree. The attribute equations of the abstract grammar define the semantics of the language.

- Second, semantic functions are redundant. In higher-order attribute grammars every computation can be modelled through attribution rules. More specifically, inductive semantic functions can be replaced by higher-order attributes. For example, a typical application of higher-order attributes is to model the (recursive) lookup function in an environment. Consequently, there is no need to have a different notation

(or language) to define semantic functions in AGs. Moreover, because we express inductive functions by attributes and attribute equations, the termination of such functions is statically checked by standard AG techniques (*e.g.*, the circularity test).

- The third characteristic is that part of the abstract tree can be used directly as a value within a semantic equation. That is, grammar symbols can be moved from the syntactic domain to the semantic domain.
- Finally, as we will describe in this paper, attribute grammar components can be "glued" via higher-order attributes.

These characteristics make higher-order attribute grammars particularly suitable to model language-based tools [TC90, Pen94, KS98, Sar99].

2.1 The Block Language

Consider a very simple language that deals with the scope rules of a block structured language: a definition of an identifier `x` is visible in the smallest enclosing block, with the exception of local blocks that also contain a definition of `x`. In the latter case, the definition of `x` in the local scope hides the definition in the global one.

We shall analyse these scope rules via our favorite (toy) language: the BLOCK language. One sentence in BLOCK consists of a *block*, and a block is a (possibly empty) list of *statements*. A statement is one of the following three things: a *declaration* of an identifier (such as `decl` a), the *use* of an identifier (such as `use` a), or a nested *block*. Statements are separated by the punctuation symbol ";" and blocks are surrounded by square brackets. A concrete sentence in this language looks as follows:

```
sentence = [ use x ; use y ; decl x ;
             [ decl y ; use y ; use w ] ;
             decl y ; decl x
           ]
```

This language does not require that declarations of identifiers occur before their first use. Note that this is the case in the first two applied occurrences of `x` and `y`: they refer to their (latter) definitions on the outermost block. Note also that the local block defines a second identifier `y`. Consequently, the second applied occurrence of `y` (in the local block) refers to the inner definition and not to the outer definition. In a block, however, an identifier may be declared once, at the most. So, the second definition of identifier `x` in the outermost block is invalid. Furthermore, the BLOCK language requires that only defined identifiers may be used. As a result, the applied occurrence of `w` in the local block is invalid, since `w` has no binding occurrence at all.

We aim to develop a program that analyses BLOCK programs and computes a list containing the identifiers which do not obey to the rules of the language. Thus, this program, called `block`, is a static semantic analyzer for the BLOCK language. It has the following type: `block` :: *Prog* → `[`*Name*`]` , where *Name* is the type of the BLOCK identifiers.

In order to make the problem more interesting, and also to make it easier to detect which identifiers are being incorrectly used in a BLOCK program, we require that the list of invalid identifiers follows the sequential structure of the input program. Thus, the semantic meaning of processing the example sentence is `[w,x]`, *i.e.*: `block` sentence = `[w,x]` .

2.2 The Attribute Grammar for the Block Language

In this section we shall describe the program `block` in the traditional attribute grammar paradigm. To define the structure of the BLOCK language we introduce two context-free grammars: one defining the concrete structure of BLOCK sentences and the other the abstract structure. We use the following notation: productions are labelled with a name for future references. To distinguish between both grammars we use lower (capital) letter for the concrete (abstract) grammar, respectively. As usual in AGs we distinguish two classes of terminal symbols: the *literal symbols* (*e.g.*, `':'`, `'decl'`, etc) which do not play a role in the attribution rules and the *pseudo terminal symbols* (*e.g.*, *Name*), which are non-terminal symbols for which the productions are implicit (traditionally provided by an external lexical analyser).

```
prog   =  root       '[' stats ']'
stats  =  onests     lststs
       |  nosts
lststs =  astat      stat
       |  conslststs stat ';' lststs
stat   =  decl       'decl' Name
       |  use        'use' Name
       |  block      '[' stats ']'
```

```
Its  =  ConsIts  It Its
     |  NilIts
It   =  Decl     Name
     |  Use      Name
     |  Block    Its
```

Fragment 1: The BLOCK concrete (left) and abstract context-free grammar (right).

That is to say that the two grammars do not match. The mapping between them will be defined via HAGs in Section 2.3. To specify the semantics of BLOCK we shall use the abstract grammar. Before we extend that grammar with attributes and attribute equations, let us discuss the semantics of BLOCK informally.

The BLOCK language does not force a *declare-before-use* discipline. Consequently, a conventional implementation of the required analysis naturally leads to a program that traverses each block twice: once for processing the declarations of identifiers and constructing an environment and a second time to process the uses of identifiers (using the computed environment) in order to check for the use of non-declared identifiers. The uniqueness of identifiers is checked in the first traversal: for each newly encountered identifier declaration it is checked whether that identifier has already been declared at the same lexical level. In this case, the identifier has to be added to a list reporting the detected errors. The straightforward algorithm to implement the BLOCK processor looks as follows:

1st Traversal	2nd Traversal
- *Collect the list of local definitions*	- *Use the list of definitions as the global environment*
- *Detect duplicate definitions (using the collected definitions)*	- *Detect use of non defined names* - *Combine "both" errors*

As a consequence, semantic errors resulting from duplicate definitions are computed during the first traversal and errors resulting from missing declarations, in the second one.

We associate an inherited attribute *dcli* of type *Env* to the non-terminal symbols *Its* and *It* that define a block. The inherited environment is threaded through the block in order to accumulate the local definitions and in this way synthesizes the total environment of the block. To distinguish between the same identifier declared at different levels, we use an attribute *lev* that distributes the block's level. We associate a synthesized attribute *dclo* to the non-terminal symbols *Its* and *It*, which defines the newly computed environment. The total environment of a block is passed downwards to its body in the attribute *env* in order to detect applied occurrences of undefined identifiers. Every block inherits the environment of its outer block. The exception is the outermost block: it inherits an empty environment. To synthesize the list of errors we associate the attribute *errs* to *Its* and *It*.

The static semantics of the BLOCK language are defined in the attribute grammar presented in Fragment 2. We use a *standard* AG notation: within the attribution rules of a production, different occurrences of the same symbol are denoted by distinct subscripts. Inherited (synthesized) attributes are prefixed with the down (up) arrow ↓ (↑). Pseudo terminal symbols are syntactically referenced in the AG, *i.e.*, they are used directly as values in the attribution rules. The attribution rules are written as HASKELL-like expressions. Copy rules are included in the AG specification (although there are well-known techniques to omit copy rules, in this paper, we prefer to explicitly define them). The semantic functions *mBIn* (standing for "must be in") and *mNBIn* ("must not be in") define usual symbol table lookup operations[1].

```
Its      <↓ lev : Int, ↓ dcli : Env, ↓ env : Env >
         <↑ dclo : Env, ↑ errs : Err >
Its  =   NilIts
         Its.dclo   =  Its.dcli
         Its.errs   =  []
       | ConsIts  It  Its
         It.dcli    =  Its1.dcli
         Its2.env   =  Its1.env
         It.env     =  Its1.env
         Its2.dcli  =  It.dclo
         Its1.dclo  =  Its2.dclo
         It.lev     =  Its1.lev
         Its2.lev   =  Its1.lev
         Its1.errs  =  It.errs ++ Its2.errs

It       <↓ lev : Int, ↓ dcli : Env, ↓ env : Env >
         <↑ dclo : Env, ↑ errs : Err >
It   =   Use    Name
         It.dclo    =  It.dcli
         It.errs    =  mBIn (Name, It.env)
       | Decl   Name
         It.dclo    =  (Pair Name It.lev) : It.dcli
         It.errs    =  mNBIn (Pair Name It.lev, It.dcli)
       | Block  Its
         It.dclo    =  It.dcli
         Its.dcli   =  It.env
         Its.lev    =  It.lev + 1
         Its.env    =  Its.dclo
         It.errs    =  Its.errs
```

Fragment 2: The BLOCK attribute grammar.

It is common practice in attribute grammars to use additional non-terminals and productions to define new data types and constructor types, respectively. The type *Env* and the constructor function Pair are examples of that:

[1] These inductive functions can be defined via higher-order attributes. Indeed, in the BLOCK higher-order attribute grammar presented in [Sar99], those functions are defined through higher-order attributes.

```
Tuple  =  Pair      Name  Int
Env    =  ConsEnv   Tuple Env
       |  NilEnv
Err    =  ConsErr   Name  Err
       |  NilErr
```

Note that, the type *Env* is isomorphic with non-terminal *Env*: the term constructor functions ConsEnv and NilEnv correspond to the HASKELL built-in list constructor functions : and [], respectively. We will use both notations to define and to construct value types.

Usually, inherited attributes are not associated with the root non-terminal of the grammar. However, to make the AG more readable, we introduce a root non-terminal: so, we can easily write the attribution rules specifying that the initial environment of the outermost block is empty (*i.e.*, the root is context-free) and that its lexical level is 0.

```
P       <↑ errs : Err >
P  =    RootAbs  Its
     Its.dcli  =  []
     Its.lev   =  0
     Its.env   =  Its.dclo
     P.errs    =  Its.errs
```

The above AG fragment together with Fragment 2 formally specify the static semantics of the BLOCK language. We call this AG the *plain* BLOCK *attribute grammar*. A higher-order variant mapping the concrete into the abstract syntax will be defined in section 2.3.

2.3 The Higher-Order Attribute Grammar for the Block Language

In order to show some characteristics of HAGs we shall consider a higher-order variant of the plain BLOCK attribute grammar. We define the mapping between the concrete and the abstract syntax of BLOCK within the AG formalism. Note that most attribute grammar-based systems provide a special notation [RT89, LJPR93], or use single purpose tools, *e.g.*, the *Ast* [Gro90] and the *maptool* [KW95], to map such grammars.

We shall build up the higher-order variant of the BLOCK AG by adding new fragments to the plain one. We begin by introducing a higher-order attribute, which represents the abstract tree. The type of this higher-order attribute is the type of the abstract tree, *i.e.*, *Its*. Thus, we associate our first higher-order attribute, called *ast*, to the production applied on the start symbol of the concrete grammar. Its declaration and its instantiation look as follows:

```
prog  =  root  '[' stats ']'
         ata  ast : Its
         ast  =  stats.ast
```

The above semantic equation defines the initial value of the attribute *ast* as the synthesized attribute *ast* of *stats*[2]. Next, we define the attribution rules that compute the abstract

[2]The declaration of a higher-order attribute is denoted with the keyword **ata**, which stands for *attributable attribute*, *i.e.*, an attribute that can be attributable.

tree. Since in this simple example the grammars are very similar, a single synthesized attribute suffices. Thus, we associate the synthesized attribute *ast* to the non-terminals of the concrete grammar. In this case, the attribute equations just use the constructor functions of the abstract grammar as the semantic functions that construct the abstract tree (*e.g.*, NilIts, ConsIts, Decl, Use and Block).

```
                stats    <↑ ast : Its >
stats   =       onests   lststs
                stats.ast  =  lststs.ast
              | nosts
                stats.ast  =  NilIts

                lststs       <↑ ast : Its >
lststs  =       astat        stat
                lststs.ast   =  ConsIts  stat.ast  NilIts
              | conslststs   stat ';' lststs
                lststs1.ast  =  ConsIts  stat.ast  lststs2.ast

                stat     <↑ ast : It >
stat    =       decl     'decl' Name
                stat.ast  =  Decl  Name
stat    =       use      'use' Name
                stat.ast  =  Use  Name
stat    =       block    '[' stats ']'
                stat.ast  =  Block  stats.ast
```

Fragment 3: Mapping concrete to abstract syntax.

In the HAG formalism, we have to include now the semantic equations that instantiate the inherited attribute occurrences of the higher-order attribute. The inherited attributes induced by the higher-order attribute *ast* are: *ast.dcli*, *ast.lev* and *ast.env*, while the synthesized are: *ast.dclo* and *prog.errs*. Those are the attributes associated to the non-terminal symbol *Its*: the type of *ast*.

```
              prog   <↑ errs : Err >
prog    =     root   '[' stats ']'
              ata  ast : Its
              ast          =  stats.ast
              ast.dcli     =  []
              ast.lev      =  0
              ast.env      =  ast.dclo
              prog.errs    =  ast.errs
```

Fragment 4: Abstract syntax as one higher-order attribute.

Observe that this grammar corresponds to the fragment that defines the attribute equations of production RootAbs of the plain attribute grammar. That fragment is replaced by this one in the higher-order variant of the AG.

3 Gluing Grammar Components via Higher-Order Attribute Grammars

One of the key characteristics of higher-order attribute grammars is the fact that they allow a component-based style of programming. More precisely, higher-order attributes are used to "*glue*" AG components. The key idea in this approach is that a higher-order AG defines a higher-order attribute whose attribute equations are defined in a different component: the reused AG component. Recall that higher-order attributes are tree-valued attributes, which are themselves trees that can be decorated.

Consider, for example, that an AG component AG_1 expresses some algorithm $\mathcal{A}$ over a grammar rooted X, and suppose that we wish to express the same algorithm when defining a new grammar, say AG_2. Under the higher-order formalism this is done as follows: firstly, we define an attributable attribute, say a with type X, in the productions of AG_2 where we need to express algorithm $\mathcal{A}$. Secondly, we extend AG_2 with attributes and attribute equations which define attributes whose values are tree-valued of type X. Thus, we use the productions of AG_1 to construct such values. After that, we instantiate the higher-order attribute with such tree-value. By definition of HAGs the generated synthesized attribute occurrences of a are defined by the attribute equations of AG_1. Finally, we just have to "*plug in*" such grammar component into the whole specification, by using the usual modular approach to define AGs. That is, the two grammars are "merged" into an equivalent monolithic HAG, before they are analised. In [SS99b] we present techniques to achieve the separate analysis and compilation of AG components.

This is exactly the approach we have taken to define the processor for the BLOCK language: first, we have defined a generic AG fragment (component) expressing the static semantics of the language. Next, we have reused such component as a higher-order attribute to build the complete language processor. This technique is similar to attribute coupled grammars [GG84, LJPR93], where the output of a grammar component is a grammar (*i.e.*, a tree-valued attribute in our case) that is the input grammar of the next component.

Noticed that by expressing the gluing of AG components within the AG formalism itself, we are able to use all the standard attribute grammar techniques, *e.g.*, the efficient scheduling of computations and the static detection of circularities. For example, the inherited/synthesized attributes of the AG components can be "connected" in any order. The HAG writer does not have to be concerned with the existence of cyclic dependencies among AG components: the circularity test will detect them for her/him. Furthermore, we can use attribute grammar techniques to derive efficient implementations for the resulting HAG. For example, we can use our deforestation techniques to eliminate the intermediate trees that glue the different components [SS99a].

3.1 An Attribute Grammar Component for Pretty-Printing

This section presents a generic pretty-printing AG component[3]. This AG component is based on the processor for HTML style tables we have presented in [SAS98]. Such processor formats possibly nested HTML tables. It computes a "pretty-printed" textual (ascii) table from a HTML (table) text. An example of accepted input and the associated output is given next.

[3]Actually, attribute grammar systems provide a special notation (*i.e.*, a fixed number of combinators) to pretty-printing the syntax tree (usually called *unparsing rules*). We have omitted such rules from the BLOCK AG since, in a different way, we present them in this section.

```
<TABLE>
<TR> <TD> This   </TD> </TR>
<TR> <TD> is  </TD> <TD> a </TD> </TR>
<TR> <TD> <TABLE>
          <TR> <TD> This </TD> <TD> is </TD> </TR>
          <TR> <TD> another </TD> </TR>
          <TR> <TD> table </TD> </TR>
          </TABLE>
</TD> <TD> table </TD> </TR>
</TABLE>
```

```
|------------------|
|This        |     |
|------------------|
|is          |a    |
|------------------|
||----------||table|
||This   |is||     |
||----------||     |
||another|  ||     |
||----------||     |
||table  |  ||     |
||----------||     |
|------------------|
```

In the computed textual table, all the lines have the same number of columns and the columns have the same length. Both features are not required in the HTML language[4]. The abstract structure of (HTML) nested tables is defined by the following abstract grammar:

```
Table  =  RootTable  Rows
Rows   =  ConsRow    Row  Rows
       |  NoRow
Row    =  OneRow     Elems
Elems  =  ConsElem   Elem  Elems
       |  NoElem
Elem   =  SName      String
       |  NestedT    Table
```

Fragment 5: The abstract grammar for nested tables.

In [SAS98], we have defined the attribute grammar over the abstract grammar and not over the HTML concrete grammar. As a result, we may use this attribute grammar component in several contexts where we need to format (abstract) tables and not to process HTML style tables only. Indeed, we wish to use this AG as a *generic pretty-printing AG component*. More recently we have extended our original AG in order to compute a LaTeX, a Xml, a Vrml, and Html table representation. Thus, we have a representation for abstract tables in all these concrete languages. We omit here the attribute grammar rules defining the pretty-printing for two reasons: firstly, they are not relevant for this paper, since we are interested in a component-back-box reuse, and secondly, we do not need to see/know them to be able to reuse the component. We only need to know the abstract grammar and the *interface*, *i.e.*, the inherited and synthesized attributes of component root symbol. This grammar component is context-free (it does not have any inherited attributes) and synthesizes all the mentioned representations: one per synthesized attribute of the root's symbol.

$$Table \quad <\uparrow ascii : String, \uparrow html : Table, \uparrow xml : Xml, \uparrow latex : Table, \uparrow vrml : Vrml >$$

[4]It is easy to see that the processor performs two traversals over the abstract tree. First, it computes the maximal height and width of each row and column, respectively. Then, it passes such values down in the tree to add "glue" where needed. Things get a bit more complicated with the nesting of the tables. As we have shown in [SAS98] it is complex to hand-write this language processor.

Although the attribute *html* and *latex* have the same type, *i.e.*, *Table*, they define different tables: while the attribute *html* does not require that all the rows have the same number of columns, the attribute *latex*, which computes LaTeX tables expressed by the tabular environment, does require it. Actually, the table synthesized in *latex* is a higher-order attribute used as an intermediate data structure to express the pretty-printing algorithm: this algorithm is easier to express if the table has a uniform number of columns. That is to say that the pretty-printing AG is itself a higher-order attribute grammar: first it builds a more suitable data structure (*i.e.*, the table with the same number of columns), and then it expresses the algorithm on that structure.

Let us now glue the pretty-printing into the BLOCK HAG. A BLOCK program can be viewed as sequence of lines (rows), each of which consists of a single element (column): a declaration, or, a use statement. Nested blocks can be viewed as nested tables. Thus, we add a new fragment to the BLOCK HAG in order to synthesize a table representation of a sentence. So, we declare a new synthesized attribute, named *table*. The additional equations simply use the constructors (productions) of the grammar component to define the abstract table[5].

```
Its      <↑ table : Rows >
Its  =   NilIts
         Its.table  =  NoRow
       | ConsIts  It  Its
         Its1.table  =  ConsRow It1.table Its2.table
It       <↑ table : Row >
It   =   Use      Name
         It.table  =  OneRow (ConsElem (SName ("use" ++ Name)) NoElem)
       | Decl     Name
         It.table  =  OneRow (ConsElem (SName ("decl" ++ Name)) NoElem)
       | Block    Its
         It.table  =  OneRow (ConsElem (NestedT (RootTable Its.table)) NoElem)
```

Fragment 6: Constructing the abstract table.

In order to reuse the table formatter AG, we have to glue it to the BLOCK AG. We glue both grammars by instantiating the higher-order attribute with the table defined by the previous fragment. Such higher-order is decorated according to the attribution rules defined in the AG component. The instantiation of the attributable attribute is defined as follows:

```
            prog   <↑ vrml : Vrml >
prog   =    root   '[' stats ']'
          ata  table : Table
          table        =  RootTable  ast.table
          prog.vrml    =  table.vrml
```

Recall that *ast* is a higher-order attribute with type *Its* that defines the abstract syntax tree (see Fragment 4). The attribute *table* is one of its synthesized attributes, as defined in

[5]We explicitly use the constructor functions induced by the abstract table grammar. We could, instead, define a set of combinator functions to construct the table using a "*concrete*" language.

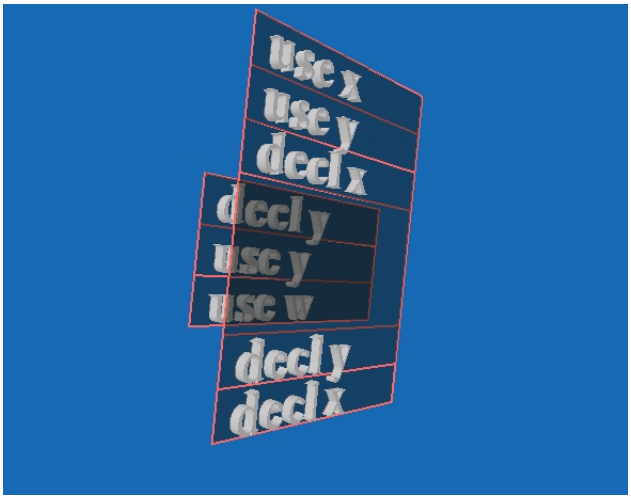

Figure 1: The 3D pretty-printed BLOCK's example sentence.

the previous fragments. Figure 1 shows the VRML representation of the example sentence. In a VRML browser an inner table appears in a different dimension than its outer one.

3.2 An Attribute Grammar Component for Graphical User Interface

As it was previously stated, types can be defined within the attribute grammar formalism. Roughly speaking, non-terminals define tree type constructors and productions define value type constructors. So, we may use this approach to introduce a type that defines an abstract representation of the interface of language-based tools. In other words, we use an abstract grammar to define an abstract interface. The productions of such a grammar represent "standard" graphical user interface objects, like menus, buttons, etc. Next, we present the so-called *abstract interface grammar*.

Visuals → Visuals0
| Visuals2 *Toplevel* *Visuals*

Toplevel → Toplevel *Frame* *String* *String*

Frames → Frames0
| Frames2 *Frame* *Frames*

Frame → Label *String*
| ListBox *Entrylist*
| PullDownMenu *String* *Menuentrylist*
| PushButton *String*
| Unparse *Ptr*
| HList *Frames*
| VList *Frames*

The non-terminal *Visual* defines the type of the abstract interface of the tool: it is a list of Toplevel objects, that may be displayed in different windows. A Toplevel construct displays a frame in a window. It has three arguments: the frame, a name (for future references) and the window title. The productions applied to non-terminal *Frame* define concrete visual objects. For example, production PushButton represents a *push-button*, production ListBox represents a *list box*, etc.

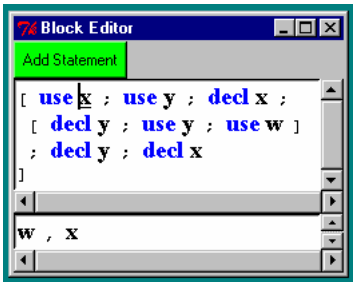

Figure 2: The BLOCK environment's interface generated from the HAG.

The production Unparse represents a visual object that provides *structured text editing* [RT89]. It displays a pretty-printed version of its (tree) argument and allows the user to interact with it (*e.g.*, to edit it). Those productions are the building blocks of the user interface. The productions VList and HList define combinators: they vertically and horizontally (respectively) combine visual objects into more complicated ones. These non-terminals and productions can be directly used in the attribute grammar to define the interface of the environments, exactly as we have done for pretty-printing in the previous section. Thus, the interface is specified through attribution, *i.e.*, within the AG formalism.

The above fragment defines the abstract structure of the interface only. To have a concrete interface, however, we have to map such abstract interface into a concrete one. Rather than defining a concrete interface from scratch and implementing a library for graphical I/O (and reinventing the wheel!), we can synthesize a concrete interface for existing high quality GUI toolkits, *e.g.*, the TCL/TK GUI toolkit [Ous94]. Indeed, the GUI AG component synthesizes TCL/TK code defining the interface in the attribute named *tk*.

Next, we present an attribute grammar fragment that glues the BLOCK HAG with this GUI AG component. It defines an interactive interface consisting of three visual objects that are vertically combined, namely: a push-button, the unparsing of the input under consideration and the unparsing of the list of errors. The root symbol *prog* synthesizes the TCL/TK concrete code in the attribute occurrence *concreteInterface*.

```
          prog  <↑ concreteInterface : Tk >
prog  =   root  '[' stats ']'
      ata  absInterface : Visuals
      absInterface          =   let  {  button  = PushButton  "Add Statement"
                                       editor  = Unparse  &stats.ast
                                       errors  = Unparse  &prog.errs
                                       comb    = VList  button : editor : errors : Frames0
                                    }  in (Toplevel  comb "edit" "Block Editor") : Visual0
      prog.concreteInterface  =  absInterface.tk
```

Fragment 7: The BLOCK graphical user interface.

Figure 2 shows the concrete interface of the BLOCK processor.

It should be noticed that the use of the abstract interface grammar (*i.e.*, an intermediate interface representation language) makes the interface highly modular: new concrete

interactive interfaces can be "*plugged in*" into the AG system by just defining the corresponding mapping function. This approach has another important property: under an incremental attribute evaluation scheme, *the interface is incrementally computed*, like any other attribute value [Sar99, SSK00].

The PushButton constructor simply displays a push-button. To assign an *action* to the displayed button we have to define such an action. Once again we use the same technique, *i.e.*, we define an abstract grammar to describe the abstract events handled by interactive interfaces. Basically, we associate an abstract event-handler to each visual object.

```
Event  →  ButtonPress  String
       |  ListBoxSelect  Entrylist
       |  MenuSelect  String
       |  TextKeyPress  Char
```

The constructor ButtonPress is the event-handler associated with PushButton. Next, we show a possible action associated with this event-handler.

```
Its  =  NilIts
     bind  on  ButtonPress "Add Statement"
           :   Its → ConsIts (Decl "a") NilIts;
```

The *bind* expression is used to specify how user interactions are handled by the language-based environment. In this case, it simply defines that every time the push-button `"Add Statement"` is pressed, the rooted subtree *Its* is transformed into ConsIts Decl("a") NilIts. Note that this event-handler constructor is defined in the context of a NilIts production. Thus, a new declaration is added at the end of the program being edited.

4 Related Work

The work presented in this paper is closely related to attribute coupled grammars [GG84, LJPR93, CDPR98], composable attribute grammars [FMY92] and Kastens and Waite work on modularity and reusability of attribute grammars [KW94].

Attribute coupled grammars consist of a set of AG components each of which (conceptually) returns a tree-valued result that is the input for the next component. Grammars are coupled by defining attribute equations that build the required tree-valued attributes, very much like the values of higher-order attributes are defined in our approach (*e.g.*, Fragments 3 and 6). In attribute coupled grammars, however, the flow of data is strictly linear and unidirectional. In our approach the data can flow freely throughout the components, provided that no attribute depends directly nor indirectly on itself. Under our techniques such cyclic dependencies are statically detected.

In [GG84] descriptional composition is defined to eliminate the creation of the intermediate trees. That is, from the coupling attribute grammar (modules) a grammar is constructed that defines the same equations, but that eliminates the construction of the intermediate trees. The descriptional composition, however, can result in a non-absolute circular AG. Furthermore, descriptional composition does not allow the separate analysis and compilation of grammar components.

Composable attribute grammars [FMY92] use a particular grammar module for gluing AG components. Grammar modules con be analised and compiled separately. However, the gluing of the components is expressed with a special notation outside the AG formalism.

Kastens and Waite [KW94] aim at a different form of modularity. They show that a combination of notational concepts can be used to create reusable attribution modules. They also define a set of modules to express common operation on programming languages. However, such modules are not defined within the AG formalism, thus, making the maintenance, updating and understanding of such components much harder.

5 Conclusions

This paper presented techniques for writing attribute grammars under a component-based style of programming. Such techniques presented rely on the higher-order attribute grammar formalism: attribute grammar components are glued into a larger AG system through higher-order attributes.

The complete specification of the BLOCK's programming environment was presented in this paper. That specification is defined within a single declarative formalism, using a single notation: higher-order attribute grammars. Indeed, the mapping between concrete and abstract syntaxes, the parsing/unparsing, the static semantics, the semantic functions, the advanced interactive graphical user interface and the gluing of AG components are specified with attributes and their equations. By expressing all the language-based tool within HAGs we inherit all the nice features of attribute grammars, namely the static detection of circularities (and termination of the AG implementations), the static scheduling of computations, and the automatic generation of incremental implementations.

We have also presented two generic, reusable and off-the-shelf AG components for pretty-printing and graphical user interface. Such components are been used to construct powerful programming environments (*e.g.*, the BiTeX environment [Sar99]). These techniques are implemented in the LRC system. The programming environment presented in figure 1 and 2 was produced by LRC from the AG specification defined in this paper.

References

[CDPR98] Loic Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Generic Programming by Program Composition. In *Proceedings of the Workshop on Generic Programming*, pages 1–13, June 1998.

[dMBS00] Oege de Moor, Kevin Backhouse, and Doaitse Swierstra. First-Class Attribute Grammars. In D. Parigot and M. Mernik, editors, *Third Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 1–20, Ponte de Lima, Portugal, July 2000. INRIA Rocquencourt.

[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.

[GG84] Harald Ganzinger and Robert Giegerich. Attribute Coupled Grammars. In *ACM SIGPLAN '84 Symposium on Compiler Construction*, volume 19, pages 157–170, Montréal, June 1984.

[Gro90] Josef Grosch. Ast - a generator for abstract syntax tree. Research Report 15, GMD, September 1990.

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

[KS98] Matthijs Kuiper and João Saraiva. Lrc - A Generator for Incremental Language-Oriented Tools. In Kay Koskimies, editor, *7th International Conference on Compiler Construction, CC/ETAPS'98*, 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.

[KW95] Basim M. Kadhim and William M. Waite. Maptool - Mapping Between Concrete and Abstract Syntaxes. Technical report CU-CS-765-95, Department of Computer Science, University of Colorado, Boulder, February 1995.

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

[Ous94] J.K. Ousterhout. *Tcl and the Tk toolkit.* Addison Wesley, 1994.

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

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

[Sar99] João Saraiva. *Purely Functional Implementation of Attribute Grammars.* PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, December 1999. ISBN 90-393-2282-1. `ftp://ftp.cs.uu.nl/pub/RUU/CS/phdtheses/Saraiva/`.

[SAS98] Doaitse Swierstra, Pablo Azero, and João Saraiva. Designing and Implementing Combinator Languages. In Doaitse Swierstra, Pedro Henriques, and José Oliveira, editors, *Third Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 150–206. Springer-Verlag, September 1998.

[SS99a] João Saraiva and Doaitse Swierstra. Data Structure Free Compilation. In Stefan Jähnichen, editor, *8th International Conference on Compiler Construction, CC/ETAPS'99*, volume 1575 of *LNCS*, pages 1–16. Springer-Verlag, March 1999.

[SS99b] João Saraiva and Doaitse Swierstra. Generic Attribute Grammars. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 185–204, Amsterdam, The Netherlands, March 1999. INRIA Rocquencourt.

[SSK00] João Saraiva, Doaitse Swierstra, and Matthijs Kuiper. Functional Incremental Attribute Evaluation. In David Watt, editor, *9th International Conference on Compiler Construction, CC/ETAPS2000*, volume 1781 of *LNCS*, pages 279–294. Springer-Verlag, March 2000.

[TC90] Tim Teitelbaum and Richard Chapman. Higher-order attribute grammars and editing environments. In *ACM SIGPLAN'90 Conference on Principles of Programming Languages*, volume 25, pages 197–208. ACM, June 1990.

[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.