Generating Spreadsheet-Like Tools from Strong Attribute Grammars

João Saraiva[1] and Doaitse Swierstra[2]

[1] Department of Computer Science
University of Minho, Portugal
jas@di.uminho.pt
[2] Department of Computer Science
University of Utrecht, The Netherlands
swierstra@cs.uu.nl

Abstract. This paper presents techniques for the formal specification and efficient incremental implementation of spreadsheet-like tools. The spreadsheets are specified by strong attribute grammars. In this style of attribute grammar programming every single inductive computation is expressed within the attribute grammar formalism. Well-known attribute grammar techniques are used to reason about such grammars. For example, ordered scheduling algorithms can be used to statically guarantee termination of the attribute grammars and to derive efficient implementations. A strong attribute grammar for a spreadsheet is defined and the first incremental results are presented.

1 Introduction

> *"Functional programming is a good idea, but we haven't got it quite right yet. What we have been doing up to now is weak (or partial) functional programming. What we should be doing is strong (or total) functional programming - in which all computations terminate..." [Tur96]*

Attribute grammars (AG) and (lazy) functional programming are closely related, as initially pointed out by Johnsson [Joh87], Kuiper and Swierstra [KS87] and recent work at Utrecht [SA98,Sar99] and Oxford [dMPJvW99,dMBS00,VWMBK02] confirms. However, what the attribute grammar community, like its functional programming counterpart, has been doing up to now is weak attribute grammar programming. There are powerful attribute grammar techniques based on attribute dependencies which statically infer important properties of the underlying attribute grammar (for example, both the circularity test and Kastens' ordered scheduling algorithm [Kas80] infer termination properties of the AG). The fact is that it is common practice in the design of attribute grammars to define (some) inductive computations via semantic functions (to perform look-up operations in an environment, for example). Although some of these computations can be efficiently defined by semantic functions (expressed in a declarative language), by defining them outside the AG formalism, they are not analysed by

F. Pfenning and Y. Smaragdakis (Eds.): GPCE 2003, LNCS 2830, pp. 307–323, 2003.
© Springer-Verlag Berlin Heidelberg 2003

the standard attribute grammar techniques. Therefore, they are simply copied/translated to the implementation of the AG (the attribute evaluator). As a result, if the semantic functions induce non-termination, then the attribute evaluator will not terminate. To complicate the task of the AG writer, the AG systems will mislead the writer by indicating that the AG is non-circular (in the case that no circularities are induced by the attribute dependencies). But, the fact is that the attribute evaluator does not terminate due to the existence of circular definitions! Indeed, all attribute grammar-based systems [RT89,JPJ+90,GHL+92,MLAŽ02,KS98] extend the AG formalism with a (declarative) language where inductive computations can be expressed. Besides the termination issue, there is a second important reason to express all our computations within the AG formalism. The attribute grammar community has done a considerable amount of work on deriving highly optimized implementations from AGs. For example, the fusion of AG fragments, the scheduling algorithms that minimize the number of traversals of the attribute evaluator, the deforestation techniques, etc. Thus, when the inductive computations are expressed within the AG formalism, we can use such techniques to derive highly optimized implementations for these computations. This is not the case when the computations are expressed as semantic functions.

The purpose of this paper is two-fold:

- First, to introduce the style of strong attribute grammar programming. Within strong attribute grammars all inductive computations are expressed via attributes and attribute equations. In this way, we can reason about our specifications within the AG setting by using well-known techniques. To achieve this style of AG programming, we rely entirely on the higher-order extension of attribute grammars [VSK89]. Within higher-order attribute grammars (HAG) every inductive computation can be modeled through attribution rules. More specifically, inductive semantic functions can be replaced by higher-order attributes. By expressing such functions within the AG formalism, we inherit all the nice properties of AGs, namely the static detection of circularities. On the other hand, within the setting of strong attribute grammars, we limit programming to the use of primitive recursion only, albeit in an effective syntactical manner.
- Second, to show that under this style of design and implementation of attribute grammars powerful tools can be constructed, we present a strong attribute grammar for a spreadsheet-like tool. In the design of this spreadsheet, we do not rely on semantic functions nor any library functions. Actually, all inductive computations are defined through attributes and their equations. Thus, the circularity test statically guarantees that this tool terminates for all possible (finite) inputs.

  Spreadsheet-like tools heavily depend on a incremental model of re-evaluation. Efficient and elegant incremental attribute evaluators for HAGs can be obtained by using function memoization [SSK00], which provides an efficient implementation for the spreadsheet by relying entirely on the AG formalism. Furthermore, HAGs provide a component-based style of programming for AGs [Sar02]. We will use this feature of HAGs to glue useful AG components to our tool. We will consider the gluing of a table formatter component (the usual representation of spreadsheets) and a query language (for querying the database).

The style of strong attribute grammar programming can be used under any AG-based system that processes higher-order attribute grammars. We will use the *Lrc* system: an incremental, purely functional, and higher-order attribute grammar-based system [KS98]. Thus, the techniques presented in this paper are implemented in *Lrc* and our spreadsheet-like tool has been produced by this system. We also present results of the incremental behaviour of this tool. When compared with previous results of incremental attribute evaluation [SSK00], these results show that spreadsheet-like tools are a natural setting for incremental evaluation.

This paper is organized as follows: Section 2 introduces strong attribute grammars. Section 3 presents a strong attribute grammar for a spreadsheet-like tool. In Section 3.1 we extend the spreadsheet with a pretty-printing SAG component. Section 4 discusses the specification of a query language within SAG and its embedding in the running spreadsheet example. Section 5 discusses the incremental evaluation of strong attribute grammars and presents the results of the incremental behaviour of the spreadsheet. Section 6 discusses related work and Section 7 presents the conclusions.

2 Strong Attribute Grammars

Strong attribute grammars (SAG) are higher-order attribute grammars where every inductive computation is defined via (higher-order) attributes and their respective equations. Or, in other words, no recursive semantic function definition is included in the AG specification. Strong attribute grammars have four key properties:

- In the first place, the standard AG-based techniques apply directly to HAGs which implies that they also apply directly to strong attribute grammars. Since all inductive computations are specified via attribute and attribute equations, we can then use well-know attribute grammar techniques to statically detect circularities (*e.g.*, the circularity test). This is not the case in classical AGs, where a non-terminating semantic function can be included in the specification. Since standard attribute grammars do not consider semantic functions, the implementation derived for such grammars will not terminate either.
- Second, there are AG-based techniques that automatically derive efficient, highly optimized implementations from a (higher-order) attribute grammar. If we express the inductive computations outside the AG formalism, then optimization will not be applied to such semantic functions by the AG systems.
  For example, when structuring an attribute grammar, it is common practice to separate different semantic aspects into different AG fragments. Then, standard attribute grammar techniques will first fuse such fragments and statically schedule the computation (minimizing the number of tree traversals), before a set of optimized evaluator functions is generated. Consider that one expresses such semantic aspects with semantic functions, like, for example, functional programmers do. In this situation, the semantic functions are just translated to the generated attribute evaluator. These semantic functions will perform several traversals over the tree (actually, one per semantic function), while they could (probably) be tupled so that the number of traversals can be reduced (and the construction/destruction of intermediate gluing

data structures eliminated by using fusion). Such optimised functions can be obtained by applying the tupling, fusion and deforestation techniques [Wad90] which are well-known in the context of functional programming. In this case, we are relying on the compiler of the AG implementation language to perform the optimizations. However, such a compiler may not be available (if it exists at all).

- Third, the attribute grammar writer defines all the required algorithms within a single programming paradigm: the attribute grammar paradigm. No previous knowledge of a different programming paradigm (*e.g.*, functional programming) or programming language is necessary to efficiently and elegantly express these algorithms within SAG. This style of AG programming also makes the understanding, maintenance and updating of the AG specifications much simpler.
- Finally, because there is no need to support inductive semantic functions, the developers of AG-based systems do not have to include in their systems a powerful declarative language (and its complicated processor) to allow their definitions. This makes the systems simpler to construct, and easier to maintain and update.

We omit here the formal definition of higher-order attribute grammars which are the basis of strong attribute grammars. For their formal definition the reader is referred to [VSK89,Sar99] . Let us, however, briefly describe HAG. Higher-order attribute grammars extend classical attribute grammars 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.

In order to show this style of attribute grammar programming we will consider now a motivating example: the design and implementation of a spreadsheet-like tool to manage a database of students and their respective marks.

3 The Students Spreadsheet Strong Attribute Grammar

Suppose that we have a (textual) database of students registered in one course. Each student (*i.e.*, register) has several attributes such as: identification number, name, and a list of marks (pairs containing the mark identification, and the value the student got). A possible (concrete) instance of the database is presented below.

```
33333,"Ana","tm"=16,"p1"=15,"p2"=17
44444,"Eduardo","tm"=12,"p1"=13,"p2"=15
```

The first register expresses that the student with number *33333*, named *Ana* got the mark 10 as theoretical mark (tm), 15 in the first project (p1) and 17 in the second one (p2). This database/language is defined by the following context-free grammar. A production p is denoted as X_0 = P X_1 ... X_n, where the name of the production, *i.e.*, p, also indicates the term constructor function P. The type of the constructor function is P :: $X_1 \rightarrow \cdots \rightarrow X_n \rightarrow X_0$ and we say that function P takes as arguments values of type $X_1 \cdots X_n$ and returns a value of type X_0. Roughly speaking, non-terminal symbols correspond to tree type constructors, and productions correspond to value constructors. We focus on the abstract structure of the language and we ignore its syntactic sugar, *e.g.*, punctuation symbols, etc.

```
Students = ConsStuds  Stud  Students
         | NoStuds
Stud     = OneStud    Int  String  Marks
Marks    = ConsMarks  Mark  Marks
         | NoMarks
Mark     = OneMark    String  Real
```

Fragment 1: The abstract grammar for the students database.

Having a database with the information about the students marks, the natural operations we would like to perform on that database are the mapping of a given formula through all the students in order to calculate, for example, their final classification. That is to say that we wish to construct a spreadsheet-like tool. To express a formula we consider a domain specific language (DSL) very much like the desk calculator language presented in [Paa95]. A concrete example of a formula is as follows:

```
"FinalMark" = ("tm" + "pm")/2
   where "pm" = ("p1" + "p2")/2
```

To define the formula that is applied to each student we have two possibilities:

- We may use a straightforward AG approach where the formula is defined as a semantic function, written in the declarative language used to express semantic functions in the AG formalism. As a result, this semantic function will not be analyzed (to infer termination properties) nor optimized by AG techniques. In this approach, the semantic function is part of the AG specification of the tool. As a result, the formula is processed statically and not dynamically. Thus, if we wish to use a different formula, the AG has to be modified, analysed and compiled in order to produced the desired tool.
- We use a key characteristic of strong attribute grammars: as semantic functions are redundant, they are modeled via higher-order attributes. Thus, we model the formula, or, more precisely, that language of formulas as a higher-order attribute of our spreadsheet. This is done as follows: first we define a grammar describing the (abstract) structure of the language of formulas and we extend it with attributes and equations. We introduce an inherited attribute to pass the list of marks as the "argument" of the formula, and a synthesised attribute to deliver the result of "applying" the formula to its inherited/argument. After that we have to "apply" such "function" in the context of every student. To do this, we just introduce a higher-order attribute to represent the abstract formula, we instantiate its inherited attribute (with the marks of a particular student) and we use its synthesised result. So, we define a DSL for formulas as a sub-language of our spreadsheet. Note that in this case the formula is processed dynamically since it is part of the input sentence. As a result, if the spreadsheet user wishes to change the formula, he just changes the part of the input sentence where the formula is defined.

Next we discuss in detail how this latter approach can be implemented in SAG. The structure of the spreadsheet is defined through the following grammar, where non-terminal *Students* is defined in Fragment 1, and non-terminal *Formula* represents the (abstract structure) of the formula under consideration.

SpreadSheet = ROOTPROD *Formula Students*
Formula = ONEFORM *Exp Decls*

Fragment 2: The abstract grammar for the Spreadsheet-like language.

Let us assume that we have an off-the-shelf AG component whose root non-terminal is *Formula*. This non-terminal has one inherited attribute (representing a finite function mapping variable names to values) and it synthesises one attribute with the value expressed by the formula. Synthesised (inherited) attributes are prefixed with the up (down) arrow ↑ (↓).

$$Formula <\downarrow env : Env, \uparrow res : Real >$$

We omit here the attribute declaration and equations of the (trivial) definition of this component. Note, however, that in this AG component we need to express an (inductive) lookup function because we have to lookup a particular mark in the inherited environment. Obviously, we define such a lookup function in the style of strong attribute grammars via higher-order attributes. See [Sar99] (page 37) where similar inductive functions are expressed within the AG formalism. To simplify our presentation, we shall consider that this SAG component is included in our specification in order to create a monolithic AG, which is then analised and the respective implementation derived.

In order to map the formula through the database of students, we have to move the (abstract) formula to the context of every student. Thus, we use another characteristic of the HAG, the so-called *syntactic references*, meaning that 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. In our example the "syntactic" symbol *Formula* is used within an equation as follows:

SpreadSheet = ROOTPROD *Formula Students*
Students.form = *Formula*

Fragment 3: Passing the formula to the students.

Instead of defining attributes and equations to move the (abstract) formula downwards in the tree (*i.e.*, the student list) via trivial *copy rules*, we use a special notation to access a remote attribute (up in the tree). The expression {*Students.form*} refers to the local attribute *form* at the non-terminal *Students* [RT89,Hed99][1]. In our AG-based system the copy rules that are automatically generated corresponding to frequently used monadic structures: reader, writer, and state monads. The AG notation, however, is much easier to use, more aspect oriented, and does not suffer from the fact that monads do not compose well.

Now that we are able to access the formula in the desired context, *i.e.*, in production ONESTUD, we use a higher-order attribute to model the semantic function that "applies" the formula to the list of marks of a student. Note that the inherited attribute *form* is a higher-order attribute: it is a (higher-order) tree that has attributes as well. In order to access those attributes we have to use the higher-order extension to the AG formalism.

[1] See [KW94,SAS99] for a survey of special notation for common attribute propagation patterns.

This is done as follows: first, we declare a higher-order attribute, *i.e.*, attributable attribute (*ata*) named *form* of type *Formula*, to represent the formula. Then, we instantiate this attribute with the inherited global formula. After that, we instantiate the inherited attribute *env* of that *ata* with the list of marks of the student (and we use a syntactic reference once again). And finally, we access the synthesised value of the formula. We use a local attribute (*finalMark*) to store the computed value. This attribute value will be included in the pretty-printed representation of the spreadsheet (Section 3.1). In the HAG notation this is expressed as follows:

```
Student = OneStud Int Name Marks
        ata form : Formula                      -- Declaration of the ata
        local finalMark : Real                  -- Declaration of a local attr.
        form     = {Students.form}              -- Instantiation of the ata
        form.env = Marks                        -- Instantiation of the inherited attr.
        finalMark = form.res                    -- Use of the synthesised attr.
```

Fragment 4: The formula as a higher-order attribute.

Before we proceed, let us compare this higher-order style of defining such computations with the classical attribute grammar one. In the classical style we could express this inductive computation by defining a (generic) semantic function that accepts as arguments the abstract representation of the formula and the list of marks of the students. It delivers the result of applying the formula to the list of marks. We can write it as follows:

```
Student = OneStud Int Name Marks
        finalMark = evalFormula({Students.form}, Marks)
```

where *evalFormula* is the inductive function. This function has to be defined and included in the AG specification. If this semantic function is semantically equivalent to the formula AG component, then, the previous two AG fragment are semantically equivalent as well. Furthermore, this approach also supports the dynamic update of the formula, since its representation is an argument of the semantic function. But, there are two important differences between these two approaches: while in the strong AG one, the AG techniques analyse the attribute dependencies, check for termination properties and, if no circularities are induced, finally, produce an optimized implementation. In the classical AG one, the function is simply translated to the output without any analysis nor optimization. As a consequence, it can cause the non-termination of the attribute evaluator. Moreover, for more elaborate semantic functions, such as pretty printing algorithm that will be presented in Section 3.1, it can be complicated to hand-write such inductive semantic functions.

The reader familiar with the implementation of (higher-order) attribute grammars may have noticed that the AG fragment just presented above corresponds to the structure of the attribute evaluator generated by the AG techniques from the higher-order one.

3.1 A Strong Attribute Grammar for Table Pretty-Printing

Spreadsheets are usually displayed (or, in other words, pretty printed) in a table-like representation. In this section we show how a generic, off-the-shelf pretty-printing SAG

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

Fig. 1. Table formatter.

component can be plugged into our students spreadsheet so as to obtain the desired representation[2]. This SAG component is based on the processor for HTML style tables we have presented in [SAS99]. It computes a pretty-printed textual (ascii) table from a HTML (table) text. An example of accepted input and the associated output is given in Figure 1. 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[3].

The abstract structure of nested tables is defined by the abstract grammar shown in Fragment 5.

```
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.

As we did previously with the formula AG component, we omit here the attribution rules of the table formatter because they are not relevant for this paper. To reuse this

[2] Actually, attribute grammar systems provide a special domain specific language (or, in other words, a fixed number of combinators) to pretty-print the syntax tree (usually called *unparsing rules*).

[3] It is easy to see that the processor performs two traversals over the abstract tree that represents the input under consideration. 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 [SAS99] it is complex to hand-write this DSL's processor.

grammar we need to know the (above) abstract grammar and its *interface*, *i.e.*, the inherited and synthesised attributes of its root symbol, named *Table*. This table formatter grammar is context-free (it does not have inherited attributes) and synthesises the pretty printed ascii representation (attribute *ascii*). More recently we have extended our original AG in order to synthesise a LaTeX, a XML, a VRML, and HTML table representation. Thus, we have a representation for our abstract tables in all these concrete languages. So, the root symbol has an additional synthesised attribute per every mentioned representation.

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

The table formatter SAG can be efficiently and concisely embedded in the specification of our spreadsheet as follows: first, we associate new attributes and attribute equations to the non-terminals and productions, respectively, of the spreadsheet abstract grammar (Fragment 2) in order to synthesise a table representation of the spreadsheet. So, we declare a new synthesised attribute, named *table*. The attribute equations simply use the constructors (productions) of the table formatter SAG in order to define the abstract table[4]. The resulting AG fragment looks as follows:

```
Students    <↑ table : Rows >
Students = ConsStuds Stud Students
         Students1.table = ConsRow Stud.table Students2.table
         | NoStuds
         Students.table = NoRow
Stud       <↑ table : Row >
Stud     = OneStud  Int String Marks
         Stud.table = OneRow (ConsElem (SName Int.str) (ConsElem (SName String)
                                (ConsElem (SName finalMark.str) NoElem)))
```

Fragment 6: Constructing the abstract table.

where *finalMark* is the local attribute declared in Fragment 4, which defines the final mark of a student according to the formula under consideration. By $finalMark.str$ we access its string representation. To make the AG more readable, we explicitly declare the attributes and their types for each non-terminal, although such information can be inferred from the semantic equations.

Having introduced attributes and their equations to define the abstract table representation of the database, we introduce now a higher-order attribute to "apply" the pretty-printing function to that abstract representation.

```
                SpreadSheet <↑ ascii : String >
SpreadSheet = RootProd  Formula Students
            ata table : Table
            table                = RootTable Students.table
            SpreadSheet.ascii = table.ascii
```

Fragment 7: "Applying" the table formatter AG component.

[4] We explicitly use the constructor function induced by the table formatter grammar. We could, instead, define a set of functions with a more friendly user-defined syntax, like in Haskell.

3.2 The Students Spreadsheet Environment

The *Lrc* system produces a programming environment from a higher-order attribute grammar. Figure 2 displays a snapshot of a spreadsheet-tool produced by *Lrc* from a AG specified using the techniques presented in this paper. *Lrc* includes an attribute grammar component for describing interactive interfaces that is embedded in the specification using the techniques described in this paper. The GUI AG is described in detail in [Sar02]. Such an AG component has standard graphical user interface objects, like buttons that can be pressed, menus that can be selected, etc (see the main window). The second window (with name *Students ascii*) is the result that we obtain (for free) after embedding the table formatter SAG. A single view (*i.e.*, the ascii view) is shown, although the five different views (*e.g.*, HTML, LaTeX, VRML, etc) are automatically computed and can be selected in the menu *Views*. In the background, a frame contains a syntax-editor to edit the pretty-printed formula (actually, we use a list of formulas) and the student database. In this syntax-editor, the user can point to a formula and dynamically change it. The user can also point to a particular student and select it through a mouse button. Then, the information of the student is displayed in a new window, where it can be easily updated. All of these actions are modelled as (abstract syntax) tree transformations, since the *Lrc* system maintains an abstract syntax tree to represent the input under consideration. Indeed, the (pretty printed) text displayed in the environments, corresponds to the textual view of such a tree. The *Lrc* system uses incremental attribute evaluation, in order to provide immediate real-time feedback, after a user action. This will be discussed in Section 5.

4 A Strong Attribute Grammar for Querying Languages

Having specified the students database, the formula to compute their final marks and how such formula is applied to each of the students, we may wish to compute which students got good or bad results. Or, we may wish to compute the students that have a final mark greater than a given number. In other words, we would like to have some mechanism to be able to query our database.

Rather than defining a particular query language for our student database, we want to define a generic query language that can not only be used for querying this particular example, but also to query any other textual database defined within the AG formalism. That is to say that we want to define a domain specific language that can be easily embedded in any AG specification. In order to not introduce yet-another querying language, we will consider the *XQuery* language: a typed, functional language for querying XML, currently being designed by the *Xml* Query Working Group of the World-Wide Web Consortium [Dra02,Wad02]. We choose *XQuery* for two reasons: first, because attribute grammars and *Xml* technology are closely related [Bra98] (both extend the context-free grammar formalism), thus *XQuery* is indeed a suitable declarative language to express queries on AG-based language specifications. Second, a *Xml* combinator library to map abstract grammars (or trees) into *Xml* documents (or *Xml* trees) is already defined (via attribute grammars) in *Lrc*. Thus, we may re-use such a library, firstly to map the abstract grammar of the language under consideration to a *Xml* document, and then to query that *Xml* document.

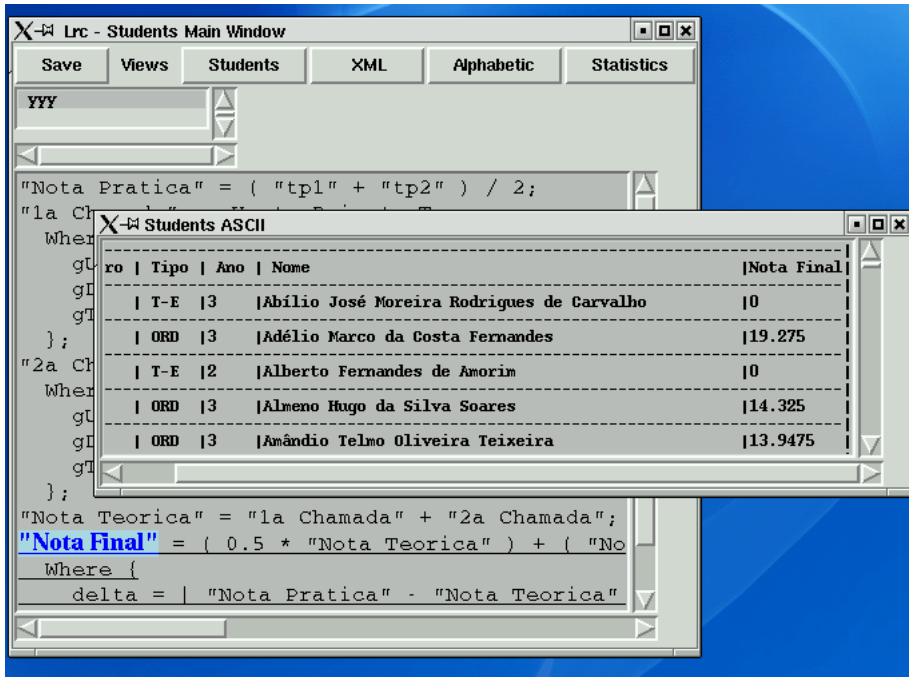


Fig. 2. A (domain specific) spreadsheet-like programming environment.

Before we briefly explain the *XQuery* language, let us present a fragment of the abstract grammar defining the structure of a *Xml* document.

```
Document = CDocument Prolog Miscs Element
Element  = CElem     Name LstAttribute LstContent
Content  = CElement  Element
         | CString   CharData
```

Fragment 8: The abstract grammar defining *Xml* documents.

XQuery uses *path expressions* that is a mechanism very much like the Unix notation to define paths on its file system. Instead of using the directory names, however, it uses the tags contained in an *Xml* document to indicate the path. Such tags correspond to the production names (or constructors) of the attribute grammar.

To introduce *XQuery*, let us consider some example queries on our student database. To list the students registered in the course we have to write the following simple *XQuery* sentence (see non-terminals and productions on fragments 1 and 2):

```
RootProd/Students
```

To list the students that got a final mark greater than `13` we can write the following query:

```
RootProd/Students/OneStud[//OneMark/@FinalMark.>.13]
```

This query selects the element `OneStud` (or the subtree constructed with constructor OneStud), that contains as descendent (the double slash `//` means that the tagged element can be a direct or an undirect descendent) an element `OneMark` where the attribute `FinalMark` is defined and it is greater than `13`.

To model *XQuery* in the SAG formalism we start by defining the (abstract) structure of this language via a context-free grammar. A fragment of such a grammar is presented next.

```
Query  = ProdQuery              AQuery
AQuery = CurrentContext         TQuery
       | RootContext            TQuery
       | DeepContextFromRoot    TQuery
TQuery = ProdBracketTQ          TQuery  XQuery
       | ProdTag                XQuery
```

Fragment 9: The abstract grammar defining *XQuery* langauge.

We extend this grammar with attributes and equations in order to synthesise the desired information, that is to say, the answer to the query under consideration. The result of a query is another *Xml* document that contains the elements of the original document that answers the query. Thus, to perform a query is to evaluate a function, say *query*, that takes the query and the *Xml* document as arguments and returns another *Xml* document. In our setting, such a function has type:

$$query :: AQuery \rightarrow Document \rightarrow Document$$

This is, once again, an inductive function that can be efficiently defined within the style of strong attribute grammar programming, using the techniques presented in Section 3. We omit here the definition of the attributes and respective equations since they are not relevant to understand our technique nor to re-use such a query language AG component.

Now that we have introduced this generic query component, we can embed it in any attribute grammar specification. For example, we can embed it in a bibliographic database processor (*e.g.*, *BibTeX* [Sar99]) to list the books written by a given author. Or, we can embed it in our spreadsheet specification in order to extend the spreadsheet environment shown in Figure 2 with a powerful querying language. Figure 3 displays a new window included in the spreadsheet that provides the user with a syntax editor to interactively query the student database. The query being displayed is the example query discussed above.

5 Incremental Evaluation of Strong Attribute Grammars

A spreadsheet-like system has to react and to provide answers in real-time. Consequently, the delay between the user interaction and the system response is an extremely important aspect in such interactive systems. Thus, one of the key features to handle such interactive environments is the ability to perform efficient re-computations. That is to say that spreadsheets must use an efficient incremental computational model. Implementing an efficient incremental engine from scratch is a complex task.

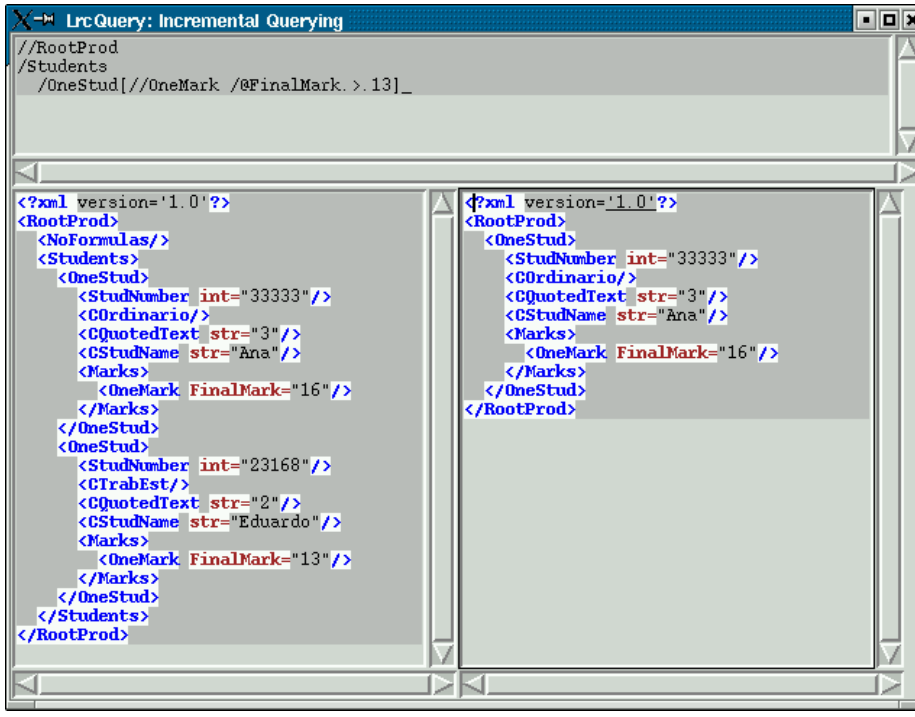


Fig. 3. Querying the students spreadsheet: the top frame displays the query that is performed on the *Xml* representation of the database (frame on the left). The answer to the query is displayed as a *Xml* document on the right frame.

A key characteristic of (higher-order) attribute grammars is that efficient incremental implementations can be automatically derived from them. A simple and efficient incremental model is obtained by standard function memoization techniques: abstract trees are collapsed into directed acyclic graphs and calls to the attribute evaluator's pure functions are memoized [SSK00].

Next, we present results obtained when executing the spreadsheet with a real student database: the database contains the students and all their marks, who followed the compiler construction course at University of Minho where this tool was proposed as the course project. The number of students attending this course was 144 and to each of them corresponds 15 evaluation elements (*i.e.*, partial evaluation marks) wich are used by a list of 7 formulas to compute different aspects of their evaluation (for example, a particular exam, the projects mark, the final mark, etc). This database was constructed and is maintained by the spreadsheet tool. Actually, we have used this tool to manage the course, as opposed to the use of a commercial tool as we had done before.

The table below presents results obtained both with non-incremental evaluation, *i.e.*, without memoization of the calls to the evaluator functions, and with incremental evaluation, *i.e.*, with memoization of the function calls. It shows the number of functions evaluated (*cache misses*), functions reused (*cache hits*), and time (in seconds on an 433

MHz Intel Celeron processor, running Linux Red-Hat 7.2). We consider five different situations: the processing of the database from scratch (*i.e.*, starting with an empty memo table), and the reaction after adding a new formula, editing one, deleting a student and editing a student mark.

	Fun. Evaluated (cache misses)	Fun. Reused (cache hits)	time (secs)
Non-Incremental:			
Scratch	217743	-	32.3
Adding a formula	229553	-	37.2
Editing a formula	217743	-	33.8
Deleting a student	216442	-	34.9
Editing a student mark	217743	-	34.8
Incremental:			
Scratch	14540	5305	5.9
Adding a formula	6375	5494	3.7
Editing a formula	2557	2131	1.9
Deleting a student	1724	1571	1.4
Editing a student mark	1652	1486	0.8

As the above table shows, our incremental model of attribute evaluation produces efficient implementations. Even when processing an input from scratch, the incremental evaluator computes 6.6% as many functions as compared to when no incrementality is used (14540 functions evaluated against 217743, respectively) and is 6 times faster. The reused functions are, in this case, due to the decoration of the same tree (representing the formula) with the same inherited attributes (the same marks). Or, in other words, the reuse of previous evaluations of the formula with the same "arguments". As expected, the tool handles very well updates of the input: adding a new (global) formula requires the re-evaluation of 0.03% of the functions computed with non-incremental evaluation and 44% of the functions if we consider incremental evaluation. Better results are obtained with local changes (*e.g.*, editing a student mark).

We have previously presented results of incremental evaluation in the context of incremental syntax-based editing [SSK00], where the performance of the incremental engine is not as efficient as the one presented here. For example, when processing the input from scratch, the incremental evaluator is 70% slower than the non-incremental one, due to the overhead of the incremental computational model. In the spreadsheet tool what happens is exactly the contrary. Indeed, these results prove that spreadsheet-like tools are a natural setting for incremental evaluation.

6 Related Work

The approach presented in this paper was strongly inspired by David Turner's work on elementary strong functional programming [Tur96] and the Charity programming language [CF92]. Strong functional programming, however, is known to be restrictive in the recursion patterns it allows. Our notion of strong attribute grammars is more flexible: we have the full expressiveness of higher-order attribute grammars. Charity

is a polymorphic, strongly-typed language. Programs in this language are built as the composition of combinators (such as anamorphisms, catamorphisms, hylomorphisms, etc). Written in this functional style, any program has a guarantee to terminate, in the sense that it can always be reduced to a head normal form, and, therefore, a result is produced for every (finite) input.

Incremental attribute evaluation can also be achieved by using Thomas Reps' *change propagation algorithm* [RTD83]. Such an algorithm stores attribute values in the nodes of the tree and propagates changes through the tree, re-using the values of non-affected attributes and computing the values of the affected ones. Roughly speaking, this algorithm memoizes results of attribute evaluation locally in the nodes where the computation is performed (while we memoize such values as results of function calls in a global memo table). Change propagation algorithms are known to give non-optimal incremental behaviour for HAG [CP96]. Our spreadsheet-like tool is a good example to show the limitations of this approach. Consider, for example, that a given formula is applied to a student, then the result of such function is memoized in one node of the subtree that represents that particular student. If the formula is applied to other student who has same marks, rather than re-evaluating the function we wish to reuse the previously computed result. Because the result of the previous application is memoized and available in the context of the first student, no reuse can then be achieved. That is to say that, under a change propagation algorithm, our spreadsheet will have a poor incremental behaviour. Reps' change propagation algorithm has been implemented in the synthesizer generator system [RT89]: an attribute grammar based system that generates language-based editors. Unfortunately, we are not able to present results using this approach since the synthesizer generator does not handle the class of higher-order attribute grammars.

7 Conclusions

This paper presented strong attribute grammars. In this style of attribute grammar programming every inductive function is defined within the AG formalism, by modelling it via higher-order attributes. Under this style of AG programming, well-known AG algorithms are used both to guarantee termination of the AG implementation and to derive efficient implementations. Furthermore, the use of attribute grammars has brought a nice aspected-oriented (syntactic) notation to programming with primitive recursive functions.

To prove that powerful tools can be constructed in SAG, a spreadsheet-like tool was constructed. This tool was efficient and elegantly specified within this style of attribute grammar programming. The *Lrc* system processed such a specification and derived a correct and efficient implementation. The results of incremental evaluation show that spreadsheets are a natural context for incremental evaluation. Actually, these results are much better than previous results of incremental evaluation (mainly produced in the context of syntax-based editing).

No inductive semantic function was included in the specification of the spreadsheet. Indeed, strong attribute grammar programming is just programming with attributes and their equations.

Acknowledgements

We would like to thank the students who followed the course *Processamento de Linguagens* in the scholar years 2000/2001 and 2001/2002 at University of Minho for their enthusiasm in discussing the design and implementation of the spreadsheet used as our running example. A special thanks to David Costa and António Faria for developing the query AG presented in Section 4. We also would like to thank the anonymous reviewers for their helpful comments.

References

Bra98. Neil Bradley. *The XML Companion*. Addison Wesley, 1998.

CF92. Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, University of Calgary, June 1992.

CP96. Alan Carle and Lori Pollock. On the Optimality of Change Propagation for Incremental Evaluation of Hierarchical Attribute Grammars. *ACM Transactions on Programming Languages and Systems*, 18(1):16–29, January 1996.

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.

dMPJvW99. Oege de Moor, Simon Peyton-Jones, and Eric van Wyk. Aspect-Oriented Compilers. In *Proceedings of the First International Symposium on Generative and Component-Based Software Engineering (GCSE '99)*, volume 1799 of *LNCS*. Springer-Verlag, September 1999.

Dra02. W3C Working Draft. *XQuery 1.0: An XML Query Language*, April 2002.

GHL+92. R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A Complete, Flexible Compiler Construction System. *Communications of the ACM,*, 35(2):121–131, February 1992.

Hed99. Gorel Hedin. Reference Attributed Grammars. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 153–172, Amsterdam, The Netherlands, March 1999. INRIA rocquencourt.

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

JPJ+90. Martin Jourdan, Didier Parigot, Catherine Julié, Olivier Durin, and Carole Le Bellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. In *ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, volume 25, pages 209–222. ACM, June 1990.

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

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

KS98. Matthijs Kuiper and João Saraiva. Lrc - A Generator for Incremental Language-Oriented Tools. In Kay Koskimies, editor, *7th International Conference on Compiler Construction, 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 grammars. *Acta Informatica*, 31:601–627, June 1994.

MLAŽ02. Marjan Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. Lisa: An interactive environment for programming language development. In Nigel Horspool, editor, *International Conference on Compiler Construction, CC/ETAPS'02*, volume 2304 of *LNCS*, pages 1–4. Springer-Verlag, April 2002.

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

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

RTD83. Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.

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

Sar99. João Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, December 1999.

Sar02. João Saraiva. Component-based Programming for Higher-Order Attribute Grammars. In Don Batory, Charles Consel, and Walid Taha, editors, *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering, GCSE 2002, Held as Part of the Confederation of Conferences on Principles, Logics, and Implementations of High-Level Programming Languages, PLI 2002, Pittsburgh, PA, USA, October 3-8, 2002*, volume 2487 of *LNCS*, pages 268–282. Springer-Verlag, October 2002.

SAS99. 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 1999.

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.

Tur96. D. A. Turner. Elementary strong functional programming. In R.Plasmeijer and P.Hartel, editors, *First International Symposium on Functional Programming Languages in Education*, volume 1022 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 1996.

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.

VWMBK02. E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In R. N. Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002*, volume 2304 of *Lecture Notes in Computer Science*, pages 128–142. Springer-Verlag, 2002.

Wad90. Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

Wad02. Philip Wadler. Xquery: a typed functional language for querying XML. In *Fourth Summer School on Advanced Functional Programming, Oxford*, August 2002.