Design, Implementation and Animation of Spreadsheets in the Lrc System

João Saraiva

Department of Computer Science,
University of Minho, Portugal.
e-mail: `jas@di.uminho.pt`

Abstract

This paper presents techniques for the design, implementation and animation of spreadsheet-like tools in the attribute grammar formalism. A real spreadsheet is formally specified and attribute grammar components that define user interfaces, querying languages and animations are plugged into the specification through higher-order attributes. From such a specification an incremental implementation is automatically derived and experimental results are presented.

Key words: Spreadsheets, Attribute Grammars, Incremental Evaluation, Functional Programming

1 Introduction

Attribute grammars (AG) [17] have proven to be a suitable formalism to the design and implementation of both domain specific and general purpose languages. In fact, powerful systems based on the attribute grammar formalism [23,13,2,9,20,18] have been constructed. These systems automatically produce very efficient/optimised implementations for languages specified via attribute grammars. While, in the beginning, AG-based systems were used mainly to specify and derive efficient (batch) compilers for formal languages, nowadays, AG-based systems are powerful tools that not only specify compilers, but also structured-based editors [23], programming environments [18], visual languages [15], complex pretty printing algorithms [28], program animations [20,25], etc. Furthermore, attribute grammars are also a suitable setting to express (circular) lazy programs [12,19,24,5], aspect oriented compilers [6], incremental algorithms [27], the XML technology [3], etc.

The purpose of this paper is three fold: first, to show that spreadsheet-like tools can be formally and concisely specified in the higher-order attribute grammar (HAG) formalism [30], and that well-known attribute grammar techniques can be used to reason about such formal specifications. For example, the AG circularity test can be used to statically detect circularities in the specification which may induce the non-termination of the spreadsheet. In this paper we present the specification in the attribute grammar formalism of a student spreadsheet, its interactive interface, and the specification of querying language to query the student database. Second, to show that efficient incremental implementations can be automatically derived from an attribute grammar. Spreadsheets heavily rely on a incremental computation model since they have to provide immediate feedback after user interaction. To derive incremental implementations from AG (usually called attribute evaluators) we use the *Lrc* system: a purely functional attribute grammar-based system [18]. In *Lrc* efficient incremental evaluation is obtained via function memoisation. And, finally, to show that the visualisation and animation of the (incremental) execution of the spreadsheet can be obtained from the AG specification. Such animations provide a visual debugger which allows the user, for example, to easily understand how the incremental evaluation is performed.

This paper is organised as follows: Section 2 briefly describes higher-order attribute grammars and the *Lrc* system. Section 3 models a spreadsheet within the AG formalism. AG components for defining an interactive interface, a query language and a visualisation and animation are presented. Section 4 discusses the incremental implementations derived by the *Lrc* system from attribute grammar sprecifications and presents the results of the incremental behaviour of the spreadsheet. Section 5 presents the conclusions.

2 The *Lrc* Attribute Grammar based System

The techniques presented in this paper are based on the *higher-order attribute grammar* formalism [29]. 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.

The *Lrc* system accepts as input a higher-order attribute grammar and generates purely functional implementations, the so-called attribute evaluators. *Lrc* generates both strict, multiple traversal attribute evaluators (C, HASKELL, and OCaml based attribute evaluators) and lazy attribute evaluators (expressed as circular lazy programs in HASKELL). Efficient incremental attribute evaluation is obtained via function memoization. The *Lrc* system not only produces batch tools (*e.g.*, compilers), but also programming environments. Such environments have a modern graphical user interface.

The higher-oder attribute grammar formalism and the *Lrc* system will be explained in detail in the next section where we present the HAG to specify a spreadsheet-like tool.

3 The Students Spreadsheet 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.

```
1,"Ana",tm=16,p1=15,p2=17
2,"Eduardo",tm=12,p1=13,p2=15
3,"David",tm=16,p1=15,p2=17
```

The first register expresses that the student with number *1*, named *Ana* got the mark 16 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 $X_0 = \mathrm{P}\ X_1\ \ldots\ X_n$, where the name of the production, *i.e.*, p, also indicates the term constructor function P. The type of the constructor function is $\mathrm{P} :: X_1 \to \cdots \to X_n \to X_0$ 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.

$$
\begin{array}{lll}
\mathit{Students} & = & \textsc{ConsStuds}\ \mathit{Stud}\ \mathit{Students} \\
 & | & \textsc{NoStuds} \\
\mathit{Stud} & = & \textsc{OneStud}\ \ \mathit{Int}\ \mathit{String}\ [\mathit{Mark}] \\
\mathit{Mark} & = & \textsc{OneMark}\ \ \mathit{String}\ \mathit{Real}
\end{array}
$$

Fragment 1: The abstract grammar for the students database.

We represent lists of non-terminals by using both the usual functional notation and explicit constructors.

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 [22]. 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 analysed (to infer termination properties) nor optimised 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.

- Or, we may use a key characteristic of higher-order attribute grammars: 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. Thus, we can model the formula, or, more precisely, that language of formulas as a higher-order attribute of our spreadsheet. That is, we extend our context-free grammar with new symbols and productions to define the language of formulas. Then, we associate inerited and synthesised attributes to its symbols to move context information to the formula sub-expressions and to synthesise the result of the formula.

  In the context of the specification of the student 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 the *Lrc* system. 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.

$$\begin{aligned} \mathit{SpreadSheet} &= \textsc{RootProd}\ \mathit{Formula}\ \ \mathit{Students} \\ \mathit{Formula} &= \textsc{OneForm}\ \ \ \mathit{Exp}\ \ \mathit{Decls} \end{aligned}$$

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 ↑ (↓).

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

We omit here the attribute declaration and equations of the (trivial) definition of this component. We shall consider that this AG component is included in our specification in order to create a monolithic AG, which is then analysed 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 one 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. In our example the "syntactic" symbol *Formula* is used within an equation as follows:

$$SpreadSheet = \textsc{RootProd}\ \mathit{Formula}\ \ \mathit{Students}$$

$$\mathit{Students.form} = \mathit{Formula}$$

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 $\{\mathit{Students.form}\}$ refers to the local attribute *form* at the non-terminal *Students* [23,10] [1] .

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. 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. In the higher-order attribute grammar notation this is expressed as follows:

```
Stud = OneStud Int Name [Mark]
    ata form : Formula                    -- Declaration of the ata
    local finalMark : Real           -- Declaration of a local attr.
    form      = {Students.form}           -- Instantiation of the ata
    form.env  = [Mark]-- 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

[1] See [16,28] for a survey of special notation for common attribute propagation patterns.

express this inductive computation by defining a 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:

$$Stud = \textsc{OneStud}\ Int\ \ Name\ \ [Mark]$$

$$finalMark = evalFormula(\{Students.form\}, [Mark])$$

Fragment 5: The formula as a semantic function.

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 higher-order one, the AG techniques analyse the attribute dependencies, check for termination properties and, if no circularities are induced, finally, produce an optimised implementation. In the classical attribute grammar approach, the function is simply translated to the output without any analysis nor optimisation. Thus, it can cause the non-termination of the attribute evaluator, and consequently the non-termination of the spreadsheet!

Spreadsheets are usually displayed in a table-like representation. In [28] we have presented a generic, off-the-shelf pretty-printing AG component that can be plugged into our students spreadsheet so as to obtain the desired representation[2]. This AG component is based on a processor for HTML style tables. It computes a pretty-printed textual table from a HTML (table) text. 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. Figure 1 displays the ascii representation of a pretty printed table.

3.1 The Specification of the Spreadsheet Programming Environment

As it was previously stated, types can be defined within the attribute grammar formalism via non-terminal symbols. So, we may use this approach to introduce a type that defines an abstract representation of the interface of spreadsheet-like tools (or more generally, language-based tools). In other words, we use an abstract context-free grammar to define an abstract interface. The productions (or constructors) of such a grammar represent "standard" graphical user interface objects, like menus, buttons, list boxes, pull donw menus, etc. Next, we present the so-called *Lrc abstract interface grammar*.

[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*).

$$\mathit{Visuals} = \textsc{CVisuals}\ [\mathit{Toplevel}]$$

$$\mathit{Toplevel} = \textsc{Toplevel}\ \mathit{Frame}\ \ \mathit{String}\ \ \mathit{String}$$

$$\begin{array}{lll}
\mathit{Frame} = & \textsc{Label} & \mathit{String} \\
\quad\mid & \textsc{ListBox} & \mathit{Entrylist} \\
\quad\mid & \textsc{PullDownMenu} & \mathit{String}\ \mathit{MenuList} \\
\quad\mid & \textsc{PushButton} & \mathit{String} \\
\quad\mid & \textsc{Unparse} & \mathit{Ptr} \\
\quad\mid & \textsc{HList} & [\mathit{Frame}] \\
\quad\mid & \textsc{VList} & [\mathit{Frame}]
\end{array}$$

Fragment 6: The *Lrc* abstract interface grammar.

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*, LISTBOX represents a *list box*, etc.

The production UNPARSE represents a visual object that provides *structured text editing* [23]. It displays a pretty-printed version of its (tree) argument and allows the user to interact with it. Such beautified textual representation of the abstract syntax tree is produced according to the unparse rules specified in the grammar. It also allows the user to point to the textual representation to edit it (via the keyboard), or to transform it using user defined transformations. 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. Thus, the interface of the spreadsheets is specified through attribution, *i.e.*, within the AG formalism.

To define a concrete interface, we need, as we have said above, to define the mapping from the abstract interface representation into a concrete one. Instead of defining a concrete interface from scratch, we synthesise a concrete interface for a existing GUI toolkit, *e.g.*, the TCL/TK GUI toolkit [21]. Indeed, this GUI component synthesises TCL/TK code defining the interface in the attribute named *tk*.

Next, we present an attribute grammar fragment that glues the spreadsheet HAG with this graphical user interface attribute grammar component. It defines an interactive interface consisting of two visual objects that are vertically combined, namely: a push-button and the unparsing of the input under consideration. The root symbol *Spreadsheet* synthesises the TCL/TK concrete code in the attribute occurrence *concreteInterface*.

```
SpreadSheet    <↑ concreteInterface : Tk >
SpreadSheet = ROOTPROD Formula Students
                  ata absInterface : Visuals
                  absInterface = let { button = PUSHBUTTON "XML"
                                       editor = UNPARSE & SpreadSheet
                                       comb   = VLIST [ button , editor ]
                                     } in [ TOPLEVEL comb "edit" "Students Editor" ]
                  SpreadSheet.concreteInterface = absInterface.tk
```

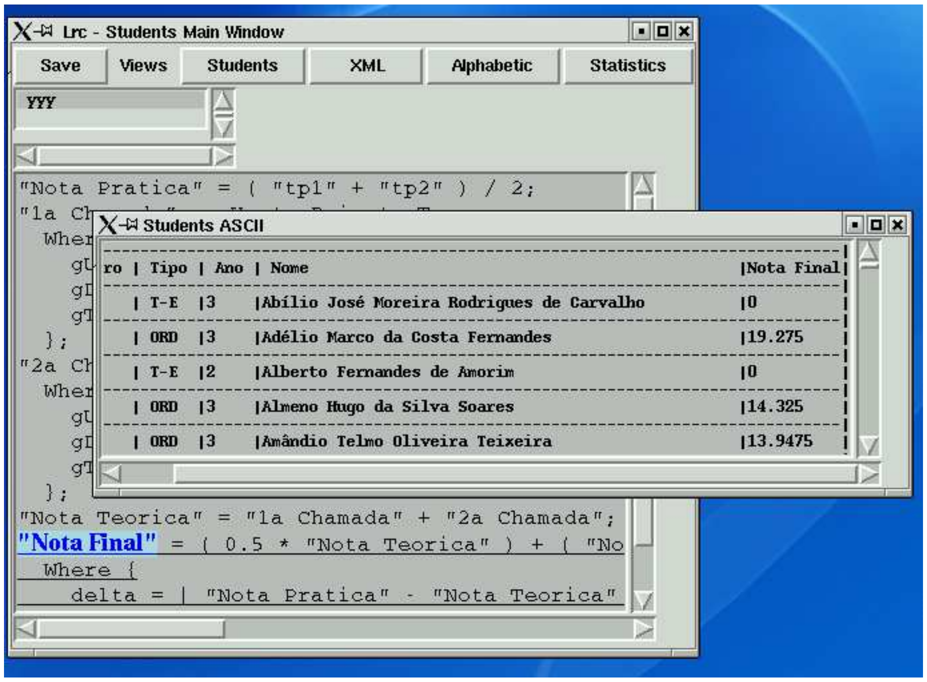
Fragment 7: The Spreadsheet graphical user interface.

Figure 1 displays a snapshot of the students spreadsheet produced by *Lrc* from a AG specified using the techniques presented in this paper. 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 4.

3.2 Querying the Spreadsheet Through Attribute Grammars

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 [7,31]. We choose *XQuery* for two reasons: first, because attribute grammars and *Xml* technology are closely related [4] (both extend the context-free grammar formalism), thus *XQuery* is indeed



```
"Nota Pratica" = ( "tp1" + "tp2" ) / 2;
"1a Ch
  Wher
    gU
    gI
    gT
  };
"2a Ch
  Wher
    gU
    gI
    gT
  };
"Nota Teorica" = "1a Chamada" + "2a Chamada";
"Nota Final" = ( 0.5 * "Nota Teorica" ) + ( "No
  Where {
    delta = | "Nota Pratica" - "Nota Teorica"
```

```
------------------------------------------------------------------------------------|
ro | Tipo | Ano | Nome                                                |Nota Final|
------------------------------------------------------------------------------------|
   | T-E  |3    |Abílio José Moreira Rodrigues de Carvalho            |0         |
------------------------------------------------------------------------------------|
   | ORD  |3    |Adélio Marco da Costa Fernandes                      |19.275    |
------------------------------------------------------------------------------------|
   | T-E  |2    |Alberto Fernandes de Amorim                          |0         |
------------------------------------------------------------------------------------|
   | ORD  |3    |Almeno Hugo da Silva Soares                          |14.325    |
------------------------------------------------------------------------------------|
   | ORD  |3    |Amândio Telmo Oliveira Teixeira                      |13.9475   |
```

Fig. 1. The spreadsheet programming environment produced by the *Lrc* system from the students spreadsheet higher-order attribute grammar.

a suitable declarative language to express queries on AG-based language specifications. Second, an *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 an *Xml* document, and then to query that *Xml* document.

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

$$
\begin{array}{lcll}
\mathit{Document} & = & \text{CD}\textsc{ocument} & \mathit{Prolog}\ \ \mathit{Miscs}\ \ \mathit{Element} \\
\mathit{Element} & = & \text{CE}\textsc{lem} & \mathit{Name}\ \ [\mathit{Attribute}]\ \ [\mathit{Content}] \\
\mathit{Content} & = & \text{CE}\textsc{lement} & \mathit{Element} \\
 & | & \text{CS}\textsc{tring} & \mathit{CharData}
\end{array}
$$

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 descendant (the double slash `//` means that the tagged element can be a direct or an indirected descendant) an element `OneMark` where the attribute `FinalMark` is defined and it is greater than `13`.

To model *XQuery* in the HAG formalism we start by defining the (abstract) structure of this language via the following abstract grammar:

$$
\begin{array}{lll}
\textit{Query} & = \textsc{ProdQuery} & \textit{AQuery} \\
\textit{AQuery} & = \textsc{CurrentContext} & \textit{TQuery} \\
 & \mid\ \textsc{RootContext} & \textit{TQuery} \\
 & \mid\ \textsc{DeepContextFromRoot} & \textit{TQuery} \\
\textit{TQuery} & = \textsc{ProdBracketTQ} & \textit{TQuery}\ \ \textit{XQuery} \\
 & \mid\ \textsc{ProdTag} & \textit{XQuery}
\end{array}
$$

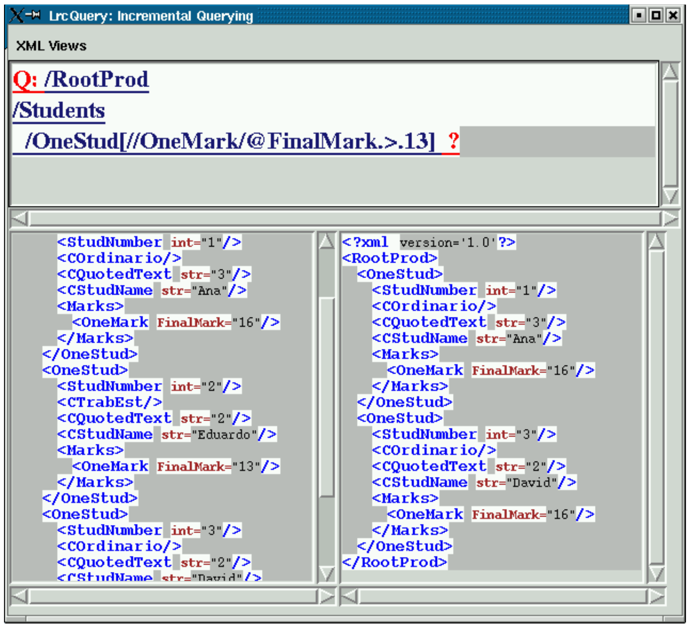
Fragment 9: The abstract grammar defining the *XQuery* language.

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:

$$query :: \textit{AQuery} \rightarrow \textit{Document} \rightarrow \textit{Document}$$

This is, once again, an inductive function that can be efficiently defined within the style of higher-order attribute grammar programming. 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* [24]) 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 1 with a powerful querying language. Figure 2 displays a new window (automatically) included in the spreadsheet that



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

```
    <StudNumber int="1"/>
    <COrdinario/>
    <CQuotedText str="3"/>
    <CStudName str="Ana"/>
    <Marks>
      <OneMark FinalMark="16"/>
    </Marks>
  </OneStud>
  <OneStud>
    <StudNumber int="2"/>
    <CTrabEst/>
    <CQuotedText str="2"/>
    <CStudName str="Eduardo"/>
    <Marks>
      <OneMark FinalMark="13"/>
    </Marks>
  </OneStud>
  <OneStud>
    <StudNumber int="3"/>
    <COrdinario/>
    <CQuotedText str="2"/>
    <CStudName str="David"/>
```

```
<?xml version='1.0'?>
<RootProd>
  <OneStud>
    <StudNumber int="1"/>
    <COrdinario/>
    <CQuotedText str="3"/>
    <CStudName str="Ana"/>
    <Marks>
      <OneMark FinalMark="16"/>
    </Marks>
  </OneStud>
  <OneStud>
    <StudNumber int="3"/>
    <COrdinario/>
    <CQuotedText str="2"/>
    <CStudName str="David"/>
    <Marks>
      <OneMark FinalMark="16"/>
    </Marks>
  </OneStud>
</RootProd>
```

Fig. 2. Querying the students spreadsheet: the top frame displays the beautified 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..

provides the user with a syntax editor to interactively query the student database. The query being displayed is the example query discussed above. Obviously, the user can dynamically modify this query and, in this case, the spreadsheet will incrementally compute the answer.

3.3 Visualisation and Animation of Spreadsheets

Attribute grammar-base systems statically schedule the computations, and, automatically generate implementations for the AGs - the so-called attribute evaluators - based on the dependencies induced by the attribution rules. Such evaluators are usually implemented as *tree-walk evaluators*: a function that walks over the abstract tree (representing the input under consideration) while computing attribute values (this task is usually called tree decoration). Thus, attribute grammars can be visualised by displaying the abstract tree in a graphical representation, and animated by displaying the tree decoration process in that tree (for example, by marking in the graphical representation the tree node being visited and by displaying the attribute values being computed).

Thus, we introduce a generic component for the visualisation and animation of AGs. We wish to use this AG as a *generic visual and animation AG component*. We start by defining an abstract grammar that is sufficiently generic to define all possible abstract tree structures we may want to visualise and animate. The grammar is as follows:

TreeViz	=	CTreeViz	*TreeId* [*TreeStmt*]
TreeStmt	=	CStmtNode	*NodeStmt*
	|	CStmtEdge	*EdgeStmt*
	|	CStmtAttr	*AttrStmt*
NodeStmt	=	CNodeStmt	*NodeId* [*Attr*]
EdgeStmt	=	CEdgeStmt	*NodeId* [*EdgeRHS*] *Attrs*
EdgeRHS	=	CRHSExpNode	*EdgeOp* *NodeId*
Attr	=	CAttr	*AttrId* *AttrVal*

The non-terminals *TreeId*, *NodeId*, *EdgeOp*, *AttrId*, *AttrVal* define sequences of characters (strings). In order to make it easier to use this component, we define a set of functions/macros that, using the productions of this AG component, define usual occurring node formats in our trees. Next, we present four functions that define the shape of a node as a record (*attrShapeRecord*), as a circle (*attrShapeCircle*), as the value of a node label (*attrLabel*), and, finally, as a node that contains a value and an arrow to a child node. These functions are presented next.

```
attrShapeRecord               =   CAttr ”shape” ”record”
attrShapeCircle               =   CAttr ”shape” ”circle”
attrLabel label               =   CAttr ”label” label
nodeRecord1 val father child  =
   [CStmtNode (CNodeStmt father) [attrShapeRecord , attrLabel (val ++ "|<c>")]
   ,CStmtEdge (CEdgeStmt "c") [CRHSExpNode "->" child] ]
```

The label is a string that defines the format of the node record. The non-terminal *EdgeOp*is a string defining the direction of the arrow.

The above grammar defines the abstract structure of abstract trees only. To have a concrete graphical representation of the trees, however, we need to map such abstract tree representation into a concrete one. Rather than defining a concrete interface from scratch and implementing a tree/graph visualization system (and reinventing the wheel!), we can synthesise a concrete interface for existing high quality graph visualization systems, *e.g.*, the GraphViz system [8]. We omit here again attributes and attribution rules that we have associated to the visualization grammar since they are neither relevant to reuse this component nor to understand our techniques.

This grammar component is context-free (it does not have any inherited attributes) and synthesises two attributes *graphviz* and *xml*, both of type string. These two attributes synthesise a textual representation of trees in the GraphViz input language. The first attribute displays trees in the usual graphic tree representation, while the second one uses a Xml tree-like representation (where the production names are the element tags).

$TreeViz \; <\uparrow graphviz : String, \uparrow xml : String>$

We are now in position to "glue" this component to the spreadsheet AG. Let us start by defining the attribute and the equations that specify the construction of the GraphViz representation.

$Students \quad <\uparrow viztree : [TreeStmt]>$

$Students = \textsc{NoStuds}$

$\quad Students.viztree = nodeEmptyCircle \; treeRef(Students)$

$\quad | \; \textsc{ConsStuds} \; Stud \; Students$

$\quad Students_1.viztree = (nodeRecord2 \; "" \; (treeRef\, Students_1) \; (treeRef\, Stud)$
$\qquad (treeRef\, Students_2)) \mathbin{++} Stud.viztree \mathbin{++} Students_2.viztree$

$Stud \quad <\uparrow viztree : [TreeStmt]>$

$Stud = \textsc{OneStud} \; Int \; String \; [Mark]$

$\quad Stud.viztree = nodeRecord1 \; ("Number : " \mathbin{++} (int2str\; Int)) \; (treeRef\, [Marks])$

Fragment 10: Constructing the Visual Tree.

Where the function *treeRef* returns a unique identifier of its tree-value argument (the tree pointer). Next, we declare a higher-order attribute, *i.e.*, attributable attribute (*ata*) named *visualTree*, in the context of the single production applied to the root non-terminal of the spreadsheet AG. The HAG fragment looks has follows:

$Spreadsheet \quad <\uparrow String : visualTree>$

$Spreadsheet = \textsc{RootProd} \; Formula \; Students$

$\quad \textbf{ata} \; visualTree : TreeViz$

$\quad visualTree = \textsc{CTreeViz} \; "Tree" \; (Formula.viztree$
$\qquad \mathbin{++} Students.viztree)$

$\quad Spreadshhet.visualTree = visualTree.graphviz$

Figure 3 shows two different snapshots (displayed by GraphViz) of the tree that is obtained as the result of running the spreadsheet with the example sentence. As we can see the tree is displayed as a *Direct Acyclic Graphs*. This happens because we are using the incremental model of attribute evaluation of *Lrc*. We will return to this subject in the next section.

Besides computing the graphical representation of the tree, the processor generated by *Lrc* also produces a sequence of node transitions. This is exactly the sequence of visits the evaluator performs to decorate the tree under consideration. Such sequence can be loaded in and animated in GraphViz, either in single step or in continuous mode, forwards and backwards. Different colors (or different shadow intensities in a black and white printing) are used to identify the number of times

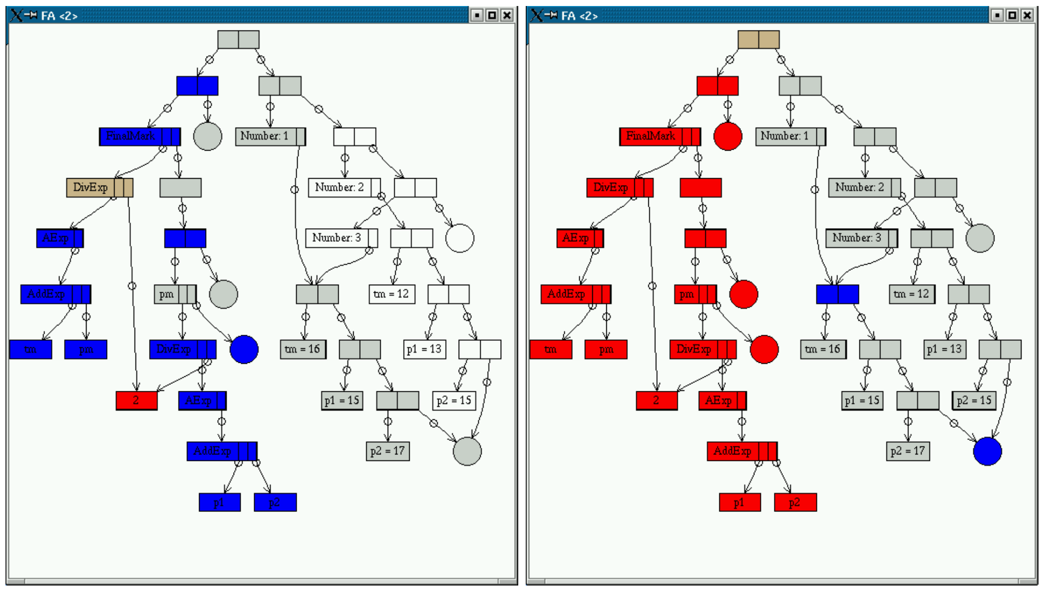


Fig. 3. The DAG representing the example sentence after processing the first student (left) and after attribute evaluation (right).

the nodes have been visited. Thus, light gray means a single visit, gray (or color blue) meand two visits and drak gray (or color red) means 3 or more visist.

The snapshot on the left shows the attribute evaluator when processing the first student. The shadowed nodes are the nodes that have already been visited. The snapshot on the right shows the tree after attribute evaluation. As we can see, the (shared) list of marks of students number 1 and 3 has been visited only once (light gray nodes). Its root, however, has been visited twice (gray node). On its second visit a cache hit occurred: the formula under consideration has already been "applied" to that argument.

4 Implementation of Attribute Grammar-based Spreadsheets

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 recomputations. That is to say that spreadsheets use an efficient incremental computational model. Implementing from scratch an efficient incremental engine is a complex task.

We use an efficient incremental computational model that efficiently (and elegantly) handles HAGs: the memoization (and posterior reuse) of calls to the functions of the attribute evaluator. Such functions traverse/visit the tree in order to assign a meaning to input under consideration. To achieve efficient function memoization we use the following combination of techniques:

- *Purely functional attribute evaluators:* Syntax trees are visited and decorated by strict, purely functional attribute evaluators. The attribute evaluators are based on the *visit-sequence* paradigm [14]: The attribute evaluator consists of a set of *visit-functions*, each of which perform the computations scheduled (by the attribute grammars scheduling algorithm) for a particular traversal of the evaluator. The attribute instances are not stored in the tree nodes, but, instead, they are the arguments and the results of pure (side-effect free) functions: the *visit-functions*. The different traversal functions are "glued" by intermediate data structures: the *visit-trees*. Such redundant intermediate structures can be eliminated by using our deforestation techniques for AGs [26].
- *Data constructor memoization:* Since attribute instances are not stored in the syntax tree, multiple instances of the syntax tree can be shared. That is, trees are collapsed into minimal Direct Acyclic Graphs (DAG) (Figure 3 displays the DAG constructed for the example input). DAGs are obtained by constructing trees bottom-up and by using constructor function memoization to eliminate replication of common sub-expressions. This technique, also called *hash-consing* [11,1], guarantees that two identical objects share the same records on the heap, and thus are represented by the same pointer.

  Data constructor memoization considerably reduces the memory usage, allows for efficient equality tests between all terms because a pointer comparison suffices and, as we will explain next, makes efficient visit-function memoization possible.
- *Visit-function memoization:* Due to the pure nature of the visit-functions, incremental evaluation is obtained by memoizing calls to the evaluator's strict visit-functions. Memoization is obtained by storing in a *function cache* calls to visit-functions. Every call corresponds to a *entry*, in the *function cache*, that records both the arguments and the results of one call to a visit-function.

  The essence of the visit-function memoization is as follows: each time a memoized visit-function is applied to a subtree and to a set of remaining arguments (*i.e.*, values of attribute instances), we search a *cache* to check whether that function was previously applied to those arguments, or not. If the cache contains an entry corresponding to the call, the result in that entry is returned. If no such entry exists, the visit-function is applied to the arguments and the call is memoized.

  In the animations produced by *Lrc* the reuse of a function call can be easily identified since when visiting a node the animation skip the visits to the children of that node. In our running example this occured in the second visit to the root of the shared list of marks.

4.1 Benchmarking the Spreadsheet

Next, we present results obtained when executing the spreadsheet with a real student database: the database contains the students, and all their marks, who follow the compiler construction course 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) which 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. 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 2.4 GHz Intel Pentium processor, running Linux Red-Hat 9). We consider six different situations: the processing of the database from scratch (*i.e.*, starting with an empty function cache), the editing of the database (*i.e.*, the reaction after adding a new formula, editing one, and editing a student mark), and the querying of the database (*i.e.*, performing a query to select a student and performing the example query).

	Non-Incremental			Incremental		
	cache misses	cache hits	time secs	cache misses	cache hits	time secs
Scratch	408568	-	7.8	41730	37052	1.65
Editing:						
Add a formula	435212	-	11.3	18356	34317	1.30
Edit a formula	408568	-	10.4	8148	11409	0.45
Edit a stud. mark	408568	-	10.4	3393	3016	0.23
Querying:						
Select a student	501208	-	12.5	22158	47412	1.63
Example query	623000	-	13.3	27871	59307	1.73

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 10% of the functions as compared to when no incrementally is used (37052 functions evaluated against 408568, respectively) and is almost 5 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 0.04% 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). A query in this spreadsheet performs like a global change since the results of applying formulas are being considered by the query. Nevertheless, using the incremental model is 8 times faster than the non-incremental one.

5 Conclusions

A spreadsheet was efficiently and elegantly specified within the 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).

References

[1] A. W. Appel and M. J. R. Gonçalves. Hash-consing Garbage Collection. Technical Report CS-TR-412-93, Princeton University, Dept. of Computer Science, Feb. 1993.

[2] L. Augusteijn. The elegant compiler generation system. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lecture Notes in Computer Science*, pages 238–254. Springer-Verlag, New York–Heidelberg–Berlin, Sept. 1990. Paris.

[3] H. Boley. Attribute grammars and xml. In *Workshop on Attribute Grammars, XML, and RDF*. University of St. Gallen, Sept. 2000.

[4] N. Bradley. *The XML Companion*. Addison Wesley, 1998.

[5] O. de Moor, K. Backhouse, and D. Swierstra. First-Class Attribute Grammars. In D. Parigot and M. Mernik, editors, *Third Workshop on Attribute Grammars and their Applications, WAGA'00*, pages 1–20, Ponte de Lima, Portugal, July 2000. INRIA Rocquencourt.

[6] O. de Moor, S. Peyton-Jones, and E. 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, Sept. 1999.

[7] W. W. Draft. *XQuery 1.0: An XML Query Language*, April 2002.

[8] E. R. Gransner and S. C. North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 00(S1):1–29, 1999.

[9] 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.

[10] G. 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, Mar. 1999. INRIA rocquencourt.

[11] J. Hughes. Lazy memo-functions. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 129–146. Springer-Verlag, September 1985.

[12] T. 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.

[13] M. Jourdan, D. Parigot, C. Julié, O. Durin, and C. 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.

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

[15] U. Kastens and C. Schmidt. Vl-eli: A generator for visual languages. In M. van den Brand and R. Laemmel, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.

[16] U. Kastens and W. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, June 1994.

[17] D. E. Knuth. Semantics of Context-free Languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).

[18] M. Kuiper and J. Saraiva. Lrc - A Generator for Incremental Language-Oriented Tools. In K. Koskimies, editor, *7th International Conference on Compiler Construction, CC/ETAPS'98*, volume 1383 of *LNCS*, pages 298–301. Springer-Verlag, April 1998.

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

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

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

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

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

[24] J. Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, December 1999. `ftp://ftp.cs.uu.nl/pub/RUU/CS/phdtheses/Saraiva/`.

[25] J. Saraiva. Component-based Programming for Higher-Order Attribute Grammars. In D. Batory, C. Consel, and W. Taha, editors, *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering, GPCE 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.

[26] J. Saraiva and D. 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, Mar. 1999.

[27] J. Saraiva, D. Swierstra, and M. 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, Mar. 2000.

[28] D. Swierstra, P. Azero, and J. Saraiva. Designing and Implementing Combinator Languages. In D. Swierstra, P. Henriques, and J. Oliveira, editors, *Third Summer*

School on Advanced Functional Programming, volume 1608 of *LNCS*, pages 150–206. Springer-Verlag, Sept. 1999.

[29] H. Vogt, D. Swierstra, and M. Kuiper. Higher order attribute grammars. In *ACM SIGPLAN ’89 Conference on Programming Language Design and Implementation*, volume 24, pages 131–145. ACM, July 1989.

[30] H. Vogt, D. Swierstra, and M. Kuiper. Efficient incremental evaluation of higher order attribute grammars. In J. Maluszynki and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 528 of *LNCS*, pages 231–242. Springer-Verlag, 1991.

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