SOCIETY FOR SCIENCE AND EDUCATION
UNITED KINGDOM

TMLAI TRANSACTIONS ON
MACHINE LEARNING AND ARTIFICIAL INTELLIGENCE

Volume 5 No. 3
ISSN 2054-7390

Functional Implementation of Multiple Traversals Program with Attribute Grammars in Scala

[1]Georges Edouard Kouamou, [2]Thierry Nenkam Nenkam, [3]Bernard Fotsing Talla
[1]National Advanced School of Engineering, University of Yaoundé 1, Cameroon;
[2]Faculty of science, University of Dschang, Cameroon;
[3]FOTSO Victor University Institute of Technology, University of Dschang, Cameroon;
thierrere@yahoo.fr; georges.kouamou@polytechnique.cm; bernard.fotsing@univ-dschang.org

ABSTRACT

Attribute grammars are a powerful specification paradigm for many language processing tasks, particularly the semantic analysis of programming languages. To functionally evaluate attributes grammar in Scala, the studies to date combine Scala with Kiama. Another study relies on object algebra, is implemented in Scala but with multiple evaluation of attributes. The purpose of this study is to use only the Scala language, in order to provide a solution to the multiple traversals of derivation trees. This approach extends the object algebra with the visit-sequences which let us increase the number of folding, thus avoid the multiple evaluation of some attributes, and the result obtained from this experience gives way to the functional modeling of interactive systems. The illustration is made on the Repmin problem.

Keywords : attribute grammars, functional programming, multi traversal program, Scala

1 Introduction

Attribute grammars was introduced by Knuth [1] especially to specify the semantic analysis of programming languages. Progressively, they were applied to a great number of systems relevant to the language processing domain, among them code generators, domain specific language implementations, refactoring tools, static code analyzers and other similar artefacts [2]. A recent work is carried out with the aim of defining a variant of attribute grammars to the modelling of distributed collaborative systems [3]. The complexity of these types of system often obliges the developers to structure their programs with multiple passes (or multiple transversals) evaluation of attributes.

The advantages in structuring a program into multiple passes address the improvement of the quality of the code which the reduction of its complexity and the increase of its reusability. Considered in isolation, each traversal (pass) may be relatively simple. Consequently, they are easier to write and are potentially more reusable. By having many distinct phases, it becomes possible to focus on a single task, rather than attempt to do many things at the same time. Furthermore, there are algorithms that rely on a multiple traversals strategy because context information must first be collected before it can be used. That is, information flows from one traversal to the next [4]. The consequence is the attribute evaluation on demand where the value of some attributes are calculated only when they are needed. If this issue is solved using the principle of lazy evaluation which is native in a pure functional language as Haskell [5], it is not the case of Scala that is a hybrid object-oriented and functional language.

DOI: 10.14738/tmlai.53.3397
Publication Date: 24th June, 2017
URL: http://dx.doi.org/10.14738/tmlai.53.3397

To face this difficulty, Sloane et al. shows how to remove the need for a generator by embedding a dynamic approach in a modern object-oriented and functional programming language [6]. But this solution combines Scala and a complementary library so-called Kiama. Similarly Object Algebra which was introduced by Bruno et al. [7], is extended to encode in Scala, the full class of L-attributed grammars that corresponds to one-pass compilers [8]. This study aims to extend the latest approach on the forms of attribute grammars which imply multiple traversals programs with the constraint to encode the solution in Scala without the use of a complementary library.

The paper is structured as follow. Section 2 introduces the notion of attribute grammar, then it presents two approaches, that we found in the literature, of functional evaluation of attribute grammars in Scala. After a synthesis on the limitations of those approaches, in section 3, the proposed approach is presented which contains an extension of the object algebra technique. Finally the paper concludes with the importance of this work on the future of our research in Section 4.

2 Scala and Attribute grammars

2.1 Attribute Grammars

A context-free grammar *G* is a quadruple: $G = (N, T, P, S)$, where *N* is a finite set of nonterminal symbols; *T* is a finite set of terminal symbols; *P* is a finite set of productions; and $S \in N$ is the distinguished start symbol of *G*. An element in $V = N \cup T$ is called a grammatical symbol. The productions in *P* are pairs of the form $X \rightarrow \alpha$, where $X \in N$ and $\alpha \in V^*$, i.e., the left-hand side X is a nonterminal symbol, and the right-hand side α is a string of grammatical symbols. An empty right-hand side (empty string) is represented by the symbol ε [2].

An attribute grammar is a context-free grammar augmented with semantic rules. More precisely, an attribute grammar is the triple $(G, Attr, R)$ where *G* is a context-free grammar, *Attr* is a finite set of all attributes and *R* a finite set of semantic rules associated to the productions of *G* [1, 2]. A fixed set of attributes is associated to each symbol of the grammar. An attribute is either synthesized or inherited. An attribute is said to be synthesized if the left hand side take the value from the right hand side. In contrast, an inherited attribute is calculated from the left hand side (the parent node) and/or siblings [9].

Example 1. The attribute grammar for the repmin problem.

The aim of the repmin problem is to construct a new tree with the same structure as the input tree, but where each leaf value is replaced by the minimum leaf value of the entire tree.

$$
\begin{array}{ll}
S \rightarrow T & T.rep = T.min \\
 & S.t = T.t \\
 & T^0.min = \min(T^1.min, T^2.min) \\
T \rightarrow TT & T^1.rep = T^0.rep \\
 & T^2.rep = T^0.rep \\
 & T^0.t = fork(T^1.t, T^2.t) \\
T \rightarrow n & T.min = n
\end{array}
$$

$$T.t = tip\ T.rep$$

“t” and “min” are synthesized attributes, “rep” is an inherited attribute for the non-terminal symbol T.

The encoding of an attribute grammar depends on the method used for the evaluation of attributes. An evaluator for a particular attribute grammar is able to traverse trees that conform to the syntax and compute the values of the attributes at each tree node.

2.2 Kiama vs Scala

Kiama is a Scala library for language processing. Scala is an interesting and powerful host language for the Kiama project due to its inclusion of both object-oriented programming and functional programming features, and its emphasis is put on scalability and interoperability with the Java virtual machine [6]. Abstract syntax trees are defined by standard Scala classes with only a minimal increase in the number of the class definitions required to prepare them for attribution. Attribute equations are written as pattern matching functions of abstract tree nodes as shown in Figure 1. Kiama provides basic synthesised and inherited attributes.

```
val locmin : Tree => Int = attr {
                        case Pair (l,r ) =>( l –>locmin ) min ( r –>locmin )
                        case Leaf ( v ) => val
                        }
val globmin : Tree => Int = attr {
                        case t if t isRoot => t->locmin
                        case t                => t.parent[tree]=>globmin
                }
val repmin : Tree => Tree = attr {
                        case Pair(l,r)   => Pair(l->repmin, r->repmin)
                        case t @ Leaf(_) => Leaf(t->globmin)
                }
```

Figure 1. Kiama solution to the Repmin problem [6]

The dynamic scheduling algorithms are the most use approaches for recent attribute grammars systems to evaluate attributes on demand. They are based on generators which complicate the development process. Sloane et al. used Kiama to remove the need for a generator by embedding a dynamic approach into a modern object-oriented and functional programming language. The result is a small, lightweight attribute grammars library that is part of a larger Kiama language processing library [6]. They use the Scala programming language because of its support for domain-specific notations and emphasis on scalability. Unlike generators with specialized notations, Kiama attribute grammars use standard Scala notations such as pattern-matching functions for equations, traits and mixins for composition and implicit parameters for forwarding.

2.3 Object Algebra

Object algebra is a programming technique that enables a simple solution to basic extensibility and modularity issues in programming languages [10]. It is an abstraction closely related to algebraic datatypes and Church encodings. Object algebras also have much in common with the traditional forms of the Visitor pattern, but without many of its drawbacks. They are extensible, they avoid the need for accept methods, and do not compromise encapsulation. To write programs with object algebras does require thinking on suitable design strategies. For example programming with traditional object-oriented visitor require the creation of generic objects and then visiting them to perform operation [11]. In object algebra, the creation of object is done relative to a factory so that specialized factories can be defined to create objects with the required operations. In brief, object algebras combine factory and visitor pattern by defining classes that implement algebraic signatures encoded as parameterized interfaces, where the type parameter represents the carrier set of the algebra. An algebraic signature defines the names and types of functions that operate over one or more abstract types, called sorts. A general algebraic signature can contain constructors that return values of the abstract set, as well as observations that return other kinds of values [9].

```
trait SigTree[-T,+T0] {
        def fork : (T, T) => T0
        def tip : Int => T0
}
```

(a) Signature of the non terminal T

```
trait SigRoot [+S, -T] {
        def rt : T => S
}
```

(b) Signature of the non-terminal S

Figure 2. Object algebra signature of the Repmin problem

Traits are a fundamental unit of code reuse in Scala. A trait encapsulates method and field definitions, which can then be reused by mixing them into classes. Unlike class inheritance, in which each class must inherit from just one superclass, a class can mix in any number of traits [10]. In *SigTree* of the Figure 2a, the methods *fork* and *tip* represent the constructors of the abstract algebra, which create values of the algebra in the carrier type *T* as type parameter of the trait. The method *tip* is defined to be an unary operation from *Int* to *T*. The trait *SigRoot* in Figure 2b has two carrier types *S* and *T*, and one constructor *rt*.

The synthesized attributes can be represented separately using a trait for each of them. But they can also be combined (composed or assembled) such that their computation is done with a single traversal. The composition is indicated to compute the attributes if they follow the same object algebra signature, otherwise they are assembled. Concerning the inherited attributes, they serve as context decorators for computation performed by synthesized attributes. This strategy is used by Rendel et al. [8] to encode the class of L-attributed grammars in the framework of object algebras.

The encoding of the above mentioned specification combines the source-language modularity properties of attribute grammars with the target-language modularity properties of object algebras in Scala. It allows referencing already computed attributes and inherited attributes can be encoded by a transformation of the algebraic signature, introducing an operation for every right hand side occurrence of a nonterminal.

2.4 Synthesis

The functional programming of attribute grammar with Kiama and Object Algebra are quite powerful. Kiama uses abstract syntax for attributable binary trees define by Scala, it provides basic synthesized and inherited attributes and semantic equations are written as pattern matching functions of abstract tree nodes. Object Algebra, introduced as a program structuring technique to improve the modularity and extensibility of programs, was studied in a relationship with the attribute grammars, in order to encode a full class of L-attributed grammars in Scala. Concerning Kiama, the interoperability with Scala is required to write multi traversal programs since the latest is used to categorize the types. Rather the examples developed in Object Algebra are relevant to a class of attribute grammars.

3 The Proposed Approach

Along this section, we continue working on the repmin problem which consists of building a new tree with the same structure as the input tree, each leaf value is replaced by the minimum value of the entire tree [12]. Remember that, the traditional solution requires a two-transversal algorithm: a first traversal to compute the minimum value of the tree, and a second one to replace all its leaf elements by the minimum value previously computed. Whichever way of composing or assembling, some attributes are evaluated more than one times.

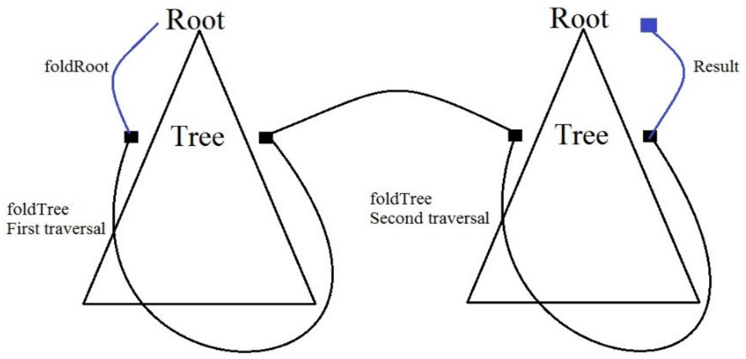

Figure 3. Traditional approach to the repmin problem.

The Kiama implementation of Figure 1 uses a common attribute propagation pattern to define the globmin attribute: defining an attribute at the root of the tree and passing it down to where it is needed [6].

Multiple traverse functional programs can be efficiently and, also, easily written within the attribute grammar formalism. The programmer does not need to concern himself with partitioning the program into a number of traversals, as in the strict functional approach. The order of evaluation is derived automatically, using standard attribute grammar techniques [4].

From the specification of the example 1, different functional programming implementations can be derived. Notice that, the circular definition of the repmin program corresponds in the attribute grammar formalism to a normal dependency from the synthesized attribute *T.min* to the inherited attribute *T.rep* of the same non-terminal symbol T. After defining multiple traversal programs within the attribute grammar formalism, we can start to encode them in Scala with object algebra by following the steps below.

- Step 1: Find the signature for synthesized attributes for every production $P: X_0 \rightarrow X_1 \dots X_n$. Find the signature for inherited attributes which will be used as context decorators for computation performed by synthesized attributes. The aim is to follow the procedure described by Rendel et al [8], for defining signatures. The signatures for Repmin previously described in figure 2 are completed with Figure 4 which presents the algebraic signature of the context decorator.

```
trait SigInhTree[-E, +T0] {
        def rt1 : E => T0
        def fork1 : T0
        def tip1 : T0
}
```

Figure 4. Algebraic signature of context decorators.

- Step 2: Define abstract computations usually called visit-sequences. They are constructed according to the following idea from Joao in his thesis [4]: a fixed sequence of abstract computations is associated to each production *P*. They abstractly describe which computations have to be performed in every visit by the evaluator to a particular type of node in the tree. Such nodes are the instances of the production P. This allows one to know the set of synthesized attributes necessary for the set of inherited attributes of P that may be used during the visit i and the set of synthesized attributes that are guaranteed to be computed by the visit *i* [8]. After that we assemble the signatures of dependent attributes in pairs of the form $(inherited\_Set, synthesized\_Set)$ for every production *P*. In the case of Repmin the result is: $synthesized\_Set = \{min, t\}$ and $inherited\_Set = \{rep\}$. The attributes "*rep*" and "*t*" are assembled below in Figure 5.

```
trait SigAssembleRep_T [ E1 , C1, E2 ] extends  TreeResult [ C1 => E1 ] {

        val  alg1 :  CtxTreeSig [ E1 , C1, E1 ]

        val  alg2 :  CtxInhTreeSig [ E2 , C1, C1 ]

}
```

Figure 5. Assembling "rep" and "t"

- Step 3: Define different "traits" for each non-terminal of the attribute grammar. Each "trait" is defined with a function "fold". And, for any dependency of the form $(inherited\_Set1, synthesized\_Set1) \rightarrow (inherited\_Set2, synthesized\_Set2)$ existing for a production, and in the "trait" of the corresponding non-terminal, we add another fold function for the computation of the synthesized values $(synthesized\_Set1)$. These values will be used in the couple $(inherited\_Set2, synthesized\_Set2)$ by the inherited values as context decorator. The illustration of this step is presented in Figure 6.

```
trait Tree {

        def fold [ E1 , C1, E2 ] ( alg1 : SigAssembleRep_T [ E1 , C1, E2 ] , Ctx : C1 ) : E1

        def Has_min [ E2 ] ( alg2 : SigTree [ E2 , E2 ] ) : E2

}

trait Root {

        def f o l d [ E1 , C1, E2 , C2 ] ( alg1 : SigRoot [ E1 , E1 ]

                , alg2 : SigAssembleRep_T [ E1 , C1, E2 ]

                        , alg3 : SigTree [ E2 , E2 ] , Ctx : C1 ) : E1

}
```

Figure 6. Complementary traits for the repmin

- Step 4: Finally, during the definition of the "cases" clauses that inherit "trait" defined above, we link the different "fold" function by using the sequences of abstract computations defined in Step 2. The sequences have already defined the order of the evaluation of the attributes. In the Repmin problem, since ”rep” serves as context decorator for the computation of ”t” once ”min” is available, the context ”rep” will be updated as shown in Figure 7. Thus “min” will be evaluated once, communicates its value to ”rep” who in return participates in the calculation of ”t”.

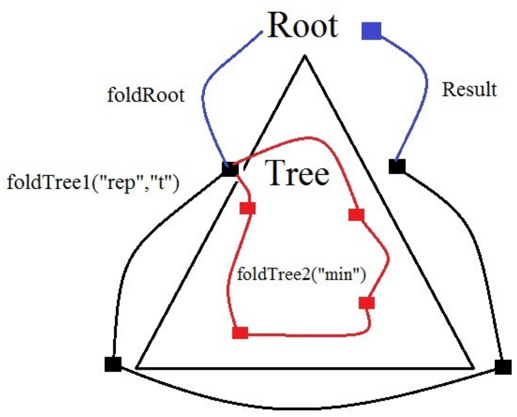

Figure 7. Illustration of the approach on the repmin problem.

4 Conclusion

The first impression, at the end of this contribution, is that Scala has allowed the implementation of multiple traversal programs specified as attribute grammars. Scala Integrates functional and imperative coding. It was used in the project Kiama for the representation of abstract trees; and in the object algebra for implementation of L-attributed grammar.

This paper presents an implementation of multiple traversal programs which extends the object algebra by increasing the number of folding with the visit-sequences. While remaining purely Scala, it shows the implementation of the Repmin problem by filling the technique of object algebra.

The attributes evaluated in a multiple traversal program as described in this paper may be available without the need to get their values through the root. Thus the intermediate values can be used once an attribute is evaluated. The Interest of this result is the use of this approach in grammar systems where the user interactions are required. In such system, user input is represented by an attribute whose the value is made available once it is acquired. In another way, we think on virtualization which is a major challenge for software engineering discipline. Software is dominated nowadays by electronic services. Considering the modelling of services composition as the production rules relevant to an attribute grammar [3], this approach can also be used to synchronize the interaction among these services.

REFERENCES

[1]. Knuth, D. E., *Semantics of Context-Free Languages*. Technical Report, California Institute of Technology, 1968.

[2]. Paakki, J., *Attribute grammar paradigms - a high-level methodology in language implementation*. ACM Computing Surveys, 1995. 2(27): p. 196-255.

[3]. Badouel, E., L. Hélouët, G.E. Kouamou, C Morvan and R. F. Nsaibirni, *Active workspaces: distributed collaborative systems based on guarded attribute grammars*. ACM SIGAPP Applied Computing Review, 2015. 15(3): p. 6-34.

[4]. Joao S., *Purely Functional Implementation of Attribute Grammars*. PhD Thesis, 1999. University of Utrecht.

[5]. Hutton G., *Programming in Haskell*. 2nd Edition, Cambridge University Press, 2016. ISBN 978-1316626221.

[6]. Sloane, A. and M. Roberts, *A pure embedding of attribute grammars*. Science of Computer Programming, 2011.

[7]. Bruno, O. and R. W., Cook, *Extensibility for the Masses Practical Extensibility with Object Algebras*. In Object Oriented Programming, 2012. Proceedings. European Conference on, Springer LNCS 7313. p. 2-27.

[8]. Rendel, T., J. I. Brachthauser and K. Ostermann, *From Object Algebras to Attribute Grammars.* In OOPSLA'14, 2014.

[9]. Johnson, T. *Attribute grammars as a functional programming paradigm*. Functional Programming Languages and Computer Architecture, 1987. Ed. FPCA, vol. 274, p. 154-173, Springer LNCS, 1987.

[10]. Bruno, O., A. Loh and R. W. Cook, *Feature-Oriented Programming with Object Algebras*. In Object-Oriented Programming, 2013. Proceedings. European Conference on, Springer LNCS 7920.

[11]. Gamma, E., R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. 1994. Addison-Wesley (USA).

[12]. Bird R. S., *Using Circular Programs to Eliminate Multiple Traversals of Data*. Acta Informatica, 1984. 21: p. 239-250.