Reference Attribute Grammars in Python

Christoffer Olsson
Lund University, Sweden
olschr2@gmail.com

Liang Ce
Lund University, Sweden
liangce0830@gmail.com

Abstract

This is a short report on a Python library that implements so called reference attribute grammars. Reference attribute grammars are in short a way to give attributes to already existing tree data structures. Attributes are computed properties of nodes in a tree, defined by equations. The ability to attribute trees is something that is very useful in for example compiler construction. There are already several implementation of reference attribute grammars in languages such as Java and Scala. However there are none in more dynamic languages such as Python. Therefore this report presents a small library written in Python that enables an end user to attribute any tree like data structure in the very same language.

1 Introduction

This is a short paper on reference attribute grammars (RAGs) implemented in Python 3. RAGs provide a way to attribute nodes in any spanning tree data structure. This is very useful in the context of compilers where the semantic analysis of a program involves the traversal of a tree. RAGs allow the compiler to assign attributes that provide semantic meaning to this intermediate representation of a program. Moreover, RAGs are generally defined outside the tree definition of the compiler, which makes them very modular by nature [1].

There already exist RAGs implementations in other languages. There is the full blown compiler construction language JastAdd, extending Java with abstract syntax tree constructions, aspect oriented programming and reference attribute grammars [2]. If one is simply interested in the ability to attribute data structures in a modular way, there are two notable implementations available, one in Java [3] and one in Scala [4].

However, all of these implementations are written in statically typed languages. This paper presents an implementation of modular and backward compatible reference attribute grammars in pure Python that attributes any spanning tree at run time. With fewer other external libraries implemented, all a user of this library needs to do in order to attribute a data structure, is to write plain Python code adhering to the library's specification.

Course paper, EDAN70, Lund University, Sweden
January 19, 2019. ACM ISBN . . . $15.00
https://doi.org/

Briefly the problems mentioned above are solved via Python's dynamic features. In Python, a class can be assigned new fields at run time, meaning that as long as the data structure exposes its nodes the library can assign new fields to them. Along with algorithms already defined in the related work. Combining these constructions, this is all one needs to define reference attribute grammars.

Since the library is written in Python, one would expect that usability and clarity to trump performance. Therefore the focus of evaluation will be on how easy and clear the library is to use. The number of lines of code needed to produce a tree and attribute it is also measured and compared to JastAdd. That is, the authors of the paper will construct a handful of programs whose sole purpose is to attribute a tree and nothing more. These programs will be used to evaluate our metrics.

It was found that the number of lines of code necessary to implement the reference attribute grammar was similar in both JastAdd and the Python library. However Python needed considerably more lines of code to actually implement the tree data structures. More on this will be elaborated upon in the section *Evaluation*.

2 Motivating Example

We will take an example, MinTree, which was implemented in JastAdd before, as illustration for this project. In the MinTree problem, an abstract syntax tree (AST) composed of node classes `Program, Pair and Leaf` are given. The class `Program` represents the root node of AST and the abstract class `Node` with two concrete subclasses, `Pair` and `Leaf`, forms the recursive tree structure. We would like to define a so called `local min` and a global min attribute. The `local min` is defined in each `Pair` as the smallest value of its children. `Global min` is defined as the smallest value defined in the whole tree. We will use RAGs to compute these values.

The main idea of JastAdd and Python solutions are the same. We firstly introduce a *synthesized attribute* for the `Node`, `localMin`, which represents the minimum value of the `Node`'s subtree. After synthesizing the information upwards in the tree, an inherited attribute, `globalMin`, is secondly introduced. Therefore, the equation of all nodes' `globalMin` is defined by an ancestor in AST, which is the Program node. Through inheriting the `globalMin` of Program node, the

minimum value of the AST is passed to all other nodes. The details are shown in Figure 2. .

Solution.png

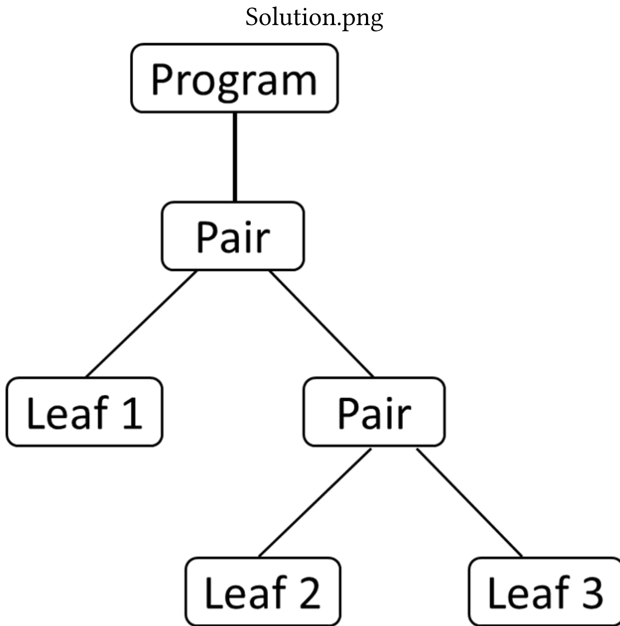

Figure 1. MinTree solution.

The following is the comparison of JastAdd and Python library. Part of the codes of Abstract Syntax Tree-specification, attribute specifications, driver codes and relevant explanations are listed below.

Abstract grammar in JastAdd:

```
Program ::= Node;
abstract Node;
Pair : Node ::= Left:Node Right:Node;
Leaf : Node ::= <Number:int>;
```

Part of abstract grammar in Python library:

```
Class Program:
 def __init__(self, child):
...
 def get_children(self):
 return self.child
```

For the abstract grammar, the tree built with JastAdd is simpler and clearer, since JastAdd has special syntax to define ASTs. JastAdd also analyzes the tree directly and identifies the parent and children of each node. In the Python library however, one needs to create classes for each node, and then also define their children and parents. The function returning node's child, getchildren(), also has to be defined for each node class.

Part of attribute specifications in JastAdd:

```
inh int Node.globalMin() = 0;
eq Program.getNode().globalMin()
= getNode().localMin();
syn int Node.localMin();
eq Node.localMin() = 0;
eq Leaf.localMin() = getNumber();
eq Pair.localMin() =
min(getLeft().localMin(), getRight().localMin()))
```

Part of attribute specification in Python library:

```
inh(Node, "globalmin")
eq(Program, 'globalmin',
lambda n: n.node.localmin())
syn(Node, "localmin")
eq(Node, "localmin", lambda n: 0)
eq(Leaf, 'localmin', lambda n: n.value)
eq(Pair, 'localmin', lambda n:
min(n.left.localmin(), n.right.localmin()))
```

The attribute specifications for JastAdd and Python are quite similar. When defining attributes in JastAdd, each attribute is declared with a type. This is not necessary in Pytohn, as the language is dynamically typed.

Driver code in Python library:

```
Weaver(MinTree)
instance = Program(Pair(Leaf(1)
, Pair(Leaf(2), Leaf(3))))
Weaver.infer_parents(instance)
```

Due to the fact that JastAdd weaves at compile time, no extra driver code is required for JastAdd. However, there is a big difference in Python. We need to use weaver as a constructor before we really implement the tree and give it access to the attribute classes we defined above. Then we have to create the instance with defining each class and the corresponding variable. The inferparents() function is a support function for inherited attributes, binding parent nodes to respective child nodes.

3 Implementation

At the current moment, the library supports synthesized and inherited attributes. For readers who are familiar with JastAdd, these work in a very similar way to how attributes in JastAdd work. However, the Python library does not support directed inheritance, which means that one can not tell an equation to only be accessible from certain children.

As stated earlier, the library is implemented in pure python. With "setattr()" and "getattr()" functions which are built in Python functions. With these one can create and manipulate class fields at run time. Therefore, due to Python's dynamic nature, both attribute calculation and assignment happen at run time, as opposed to JastAdd, where only calculation happens at run time.

In addition to these highly mutable instances, higher order functions, as well as the ability to manipulate classes in the same way as one can manipulate class instances, are exploited. As a result, fields of a class can be changed without creating an instance of it. Also, functions can be defined in some class that simply assume properties of its children. This works properly thanks to higher order functions.

Synthesized attributes are straight forward to implement. One simply passes the name of the class, the name of the attribute one wants to create and a reference to any Python function that is unevaluated. As a matter of fact, anything callable can be used as an attribute, including Python lambda expressions.

Inherited attributes are more complicated to implement. One defines an inherited attribute in a node. An inherited attribute simply needs a home class and a name. Then one defines the equation itself in another class. Later, when an instance of the tree that is to be attributed is created, one has to call a function that binds all parents using the getchildren function. When one then evaluates an inherited attribute, the library traverses the tree. It starts with the node instance where the attribute was invoked and then searches for any parent that contains an equation with the same name as the invoked attribute. When it finds one it returns the attribute reference to the starting node, meaning that as far as the user of the library is concerned, the function might as well have resided inside the node that contains the inherited attribute.

4 Evaluation

The number of lines of code to use the library was measured. That is the number of lines to specify the attribute grammar, as well as the number of lines to specify the abstract grammar. The library has no functionality to automatically construct abstract grammars and therefore that has to be done manually in the Python library. Even though this is not a feature of the library it could help the end user to assess which library he should use.

All of these results were compared to the same problems in JastAdd. The results are found in Table 1. The *Min Tree* example is the same one as in the introduction. The other two examples *Calc* and *State Machine Name Analysis* is also from the compiler course *EDAN65*.

	Python		JastAdd	
	Abs. Gramm.	Attr.	Abs. Gramm.	Attr.
Min Tree	21	10	4	12
Calc	41	14	7	8
State Machine Name analysis	25	17	5	12

Table 1. Lines of Code

In order to make these measurements the tool *CLOC* was used [5]. This is a Linux tool that simply counts the number of lines in a text file. It is tailored to work with programming languages.

A decision to not measure execution speed was also made. This was done due to a lack of time in the project. Although it would be interesting to measure the execution speed, there is no reason to believe that it would be faster than JastAdd is. This is due to the fact that JastAdd does it weaving at compile time, as opposed to at run time. Moreover native Python is in general slower than Java code.

5 Related work

In a way this project was conceived as the *dynamic variant of JastAdd*. Therefore virtually all comparisons, discussion and design was done with JastAdd in mind. The goal was more or less to simply mimic JastAdd, but in a Pythonic way [2].

The biggest difference between the Python library and JastAdd, is that JastAdd is *static* by nature. This can be illustrated in Figure 2. That is, JastAdd takes files that the

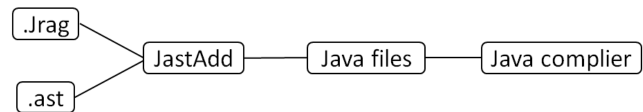

Figure 2. JastAdd Weaving.

user defines and outputs normal Java files that a Java compiler then can compile. This is different in contrast with the Python implementation which can be depicted as in Figure 3. The python implementation does everything at run time. That is all is done using the python interpreter. One simply imports the RAG library, defines the RAG grammar, as well as some tree data structure and then runs this as a normal Python program.

Since the Python library is basically a dynamic JastAdd clone it is relevant to discuss which features that are implemented at this time. JastAdd is a vast library with a wide

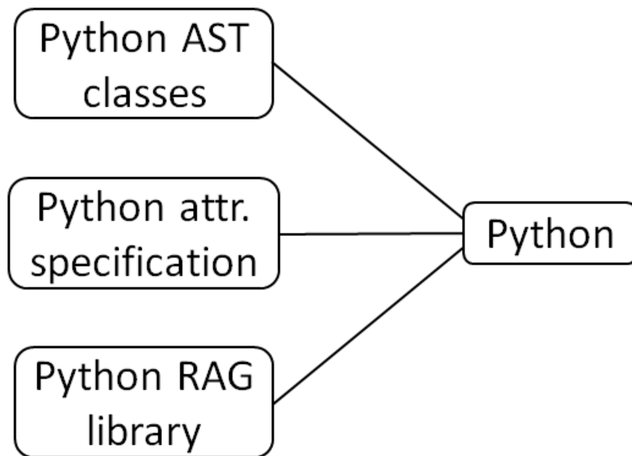

Figure 3. Python Weaving.

set of features that the reader best can explore for oneself [2].

The subset of features that are currently implemented are the following.

1. Synthesized Attributes
2. Undirected Inherited Attributes
3. Parameterized Attributes
4. Caching
5. Parent Inference
6. Attribution of any tree like data structure
7. Modularization - RAG grammar can be split into several files

Even though how these features are executed inside the computer varies between the two libraries, the aim of this project is to make them conceptually similar.

6 Conclusion and Further Work

The Python RAG library gives the end user the ability to attribute any tree like data structure in Python. JastAdd has shown that this is an incredibly useful feature, especially when designing compilers. Even though JastAdd provides a full set of tools, no RAG has been written in a dynamic language such as Python. This way it can be interesting for an end user to use the Python library if the very same user has tree like data structures in the Python language, that he wants to attribute.

This also brings us to what the next step for the library is. At this moment a small but useful sub set of JastAdd's features are implemented. A logical conclusions would be to simply implement the rest of the features that JastAdd has, starting with the most urgents ones. Which these are depends on the use case of the end user. However the authors can think of a couple of features that would be much appreciated.

In JastAdd it is possible to define ASTs in an *.ast* file. This was what made the specification of the AST classes in the evaluation section so concise in JastAdd. That would be a nice feature to have in the Python Library as well, since that would make the library more useful for end users that want to start new projects in Python.

Other than that JastAdd has more sophisticated attribute features, such as *circular attriutes, non terminal attributes, collection attributes* and so on [2]. These are all very necessary for a RAG library and the Python RAG library would benefit very much from having these as well.

Regarding the results, the Python library needed a similar amount of lines to implement the RAG grammar. Usually one gives Python praise for its ability to be a very concise and clear language. Therefore it was pivotal that the syntax of the RAG library followed this as well. Since we can't expect Python to be faster than Java, we should at least aim to make the code easy to read and write. This is an issue that is always in the eye of the beholder, but at least the number of lines needed is very similar. Therefore the Python library might actually be useful for already existing Python projects.

Also as a concluding remark, this library would benefit from a lot more bug testing. Currently the test coverage is pretty weak and would this library actually be used in real projects, the end users have to be certain that it does not contain devastating bugs.

Luckily, this library is open source under a very liberal license. The library is also currently very small in it implementation, around 100 lines of code. So any wanted features should be easy to implement at this current state of the project.

Acknowledgments

A special thanks to our project supervisor Niklas Fors. Also a special thanks to the JastAdd team, as this project would not have existed without the library.

References

[1] G. Hedin, An Introductory Tutorial on JastAdd Attribute Grammars. In: Generative and Transformational Techniques in Software Engineering III. GTTSE 2009. Lecture Notes in Computer Science, vol 6491. Springer.

[2] "JastAdd.org", Jastadd.org, 2018. [Online]. Available: http://jastadd.org/web/. [Accessed: 15- Nov- 2018].

[3] Niklas Fors, Gustav Cedersjö, and Görel Hedin. JavaRAG: a Java library for reference attribute grammars. In Proceedings of the 14th International Conference on Modularity (MODULARITY 2015). ACM, 2015, pp. 55-67.

[4] Sloane, A.M., Kats, L.C. and Visser, E. A pure embedding of attribute grammars. Science of Computer Programming, 2013, 78(10), pp.1752-1769.

[5] "CLOC – Count Lines of Code", Cloc.sourceforge.net, 2018. [Online]. Available: http://cloc.sourceforge.net/. [Accessed: 21- Dec- 2018].