Implementing Attributes in SDF

Alexandre Borghi Valentin David Akim Demaille Olivier Gournet

Epita Research and Development Laboratory (LRDE*)

Abstract

Attribute Grammars (AGs) provide a very convenient means to bind semantics to syntax. They enjoy an extensive bibliography and are used in several types of applications. Yet, to our knowledge, their use to disambiguate is novel. We present our implementation of an evaluator of attributes for *ambiguous* AGs, tailored to ambiguous parse trees disambiguation. This paper focuses on its implementation that heavily relies on Stratego/XT, which is also used as language to express the attribute rules. A companion paper presents the disambiguation process in details (David et al., 2005).

1 Introduction

In any typical compiler, or structured text crunching application, semantic passes follow the usual parser. Some of these passes are gatherers and merely compute additional information about the Abstract Syntax Tree (AST): binding, type-checking. Other passes involve modification of the AST, or even its full rewrite: desugaring, translation to another language etc. Stratego is a language of choice to express the latter kind of passes, where rewriting rules are the core of the process. ASF+SDF on the one hand, and Stratego with its dynamic rules on the other hand, will both happily help one to write annotating passes. Weirdly enough, the old and well-known AGs (Knuth, 1968) do not seem to have make it into this world, although they are very well suited to write gather-and-annotate passes.

An AG is a context-free grammar enriched with attributes bound to its symbols, and rules attached to its production rules to express the relationship between the attributes of the symbols of the rule. A very specific feature of AGs is that these local relationships suffice: their analysis reveals their dependencies, from which the order of evaluation is computed. In other words the user focuses on local issues, and the system conducts the global evaluation. Evaluation strategies range from extremely naive (repeatedly traverse the tree and compute attributes which definition uses computed attributes), to extremely smart ("compile" the AG into an evaluator which makes the best use of statically computed dependencies).

This paper presents a functional (naive) prototype supporting attributes in Syntax Definition Formalism (SDF). As a novel feature, our proposal supports *ambiguous* AGs: attributes are computed on parse *forests*. This makes it possible to express disambiguation filters thanks to attributes: a special attribute is used to flag branches that are incorrect according to semantic rules; a latter filter prunes them.

2 Current implementation

The package `sdf-attribute` is a set of tools relying on Stratego/XT to provide attributes support within SDF. It includes an extended SDF grammar to specify the syntax of attribute rules (Sec. 2.1). An extended SDF grammar is processed by a chain of tools in order (i) to check that the attribute rules are complete, and (ii) to compile the attribute rules into an evaluator (Sec. 2.2). Finally, this evaluator in run on an actual input to evaluate the attributes (Sec. 2.3).

2.1 Syntax

Attribute rules are embeded into SDF grammars via annotations (Fig. 1). In the future, a nicer syntax might be proposed. Each attribute has a name and a name space name, written `NodeName.namespace:name`. The optional name space name defaults to that of the

*`<transformers@lrde.epita.fr>`

```
1 e1:Exp "+" e2:Exp → Exp
2   {attributes(eval:
3     root.value := <add> (e1.value, e2.value)
4   )}
```

Attribute rules are (currently) regular SDF annotations under the name `attributes`. To avoid attribute name clashes, named scopes are provided (`eval` in line 2). The SDF symbol labeling feature provides convenient shorthands for symbol names, or when symbols occur several times (two `Exp`s in line 3). The special identifier `root` refers to the symbol defined, here `Exp`.

Figure 1: Example of attributes

rule list.

A rule specifies the value of an attribute via a Stratego strategy as follows: `Node.attr := strategy`. Within the strategies themselves, attributes are used like ordinary Stratego variables.

The node name is either `root` to refer to the produced non terminal, or the symbol name, or a label name. Label names are useful when a child is not a simple symbol (lists, options, etc.) or when symbol name is used several times.

2.2 Evaluator Generation

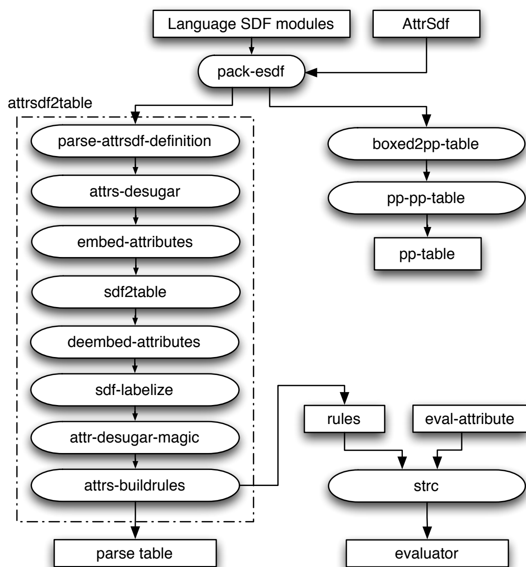

Figure 2: Parse table generation

The SDF modules are packed according to `AttrSdf`, which extend SDF with our attribute syntax. This is performed by `esdf` described in another paper (Demaille et al., 2005). Then, a parse table and an evaluator are genererated by the whole `attrsdf2table` chain, composed of several steps (Fig. 2).

Firstly, desugaring filters are run on the syntax definition to handle details such as adding implicit name spaces (`attrs-desugar`). In order to have `sdf2table` accept the Stratego code in the annotations, some transformations are applied (`embed-attributes`) and reversed afterwards in the parse table (`deembed-attributes`). Then, missing labels are inserted (`sdf-labelize`).

The next filter (`attr-desugar-magic`) addresses two important issues. First, at this stage, it is useful to check whether the attribute rules are well written or not. Since there is no debugging tool it is better to find and report errors instead of creating an invalid table. This definition checker traverses the graph of all possible trees beginning from the start symbols, carrying knowledge about attributes dependency. Second, taking advantage of this traversal, it also automatically propagates attributes which were declared inherited or synthesized, by adding the implicit rules. This is convenient symbol tables for instance.

Finally, attribute rule code is extracted from the parse table (`attr-buildrules`), and put in a rules section in a Stratego source file. It is compiled along with the tree traversal code (`eval-attribute`) to provide a filter to be used in the evaluator. This code is then erased from the parse table, to keep it as simple as possible. Only dependencies between synthesized and inherited attributes rules are kept for each node, to help the evaluation.

2.3 Evaluation

The evaluation of attributes is also performed by a chain of tools, presented in Fig. 3. SGLR

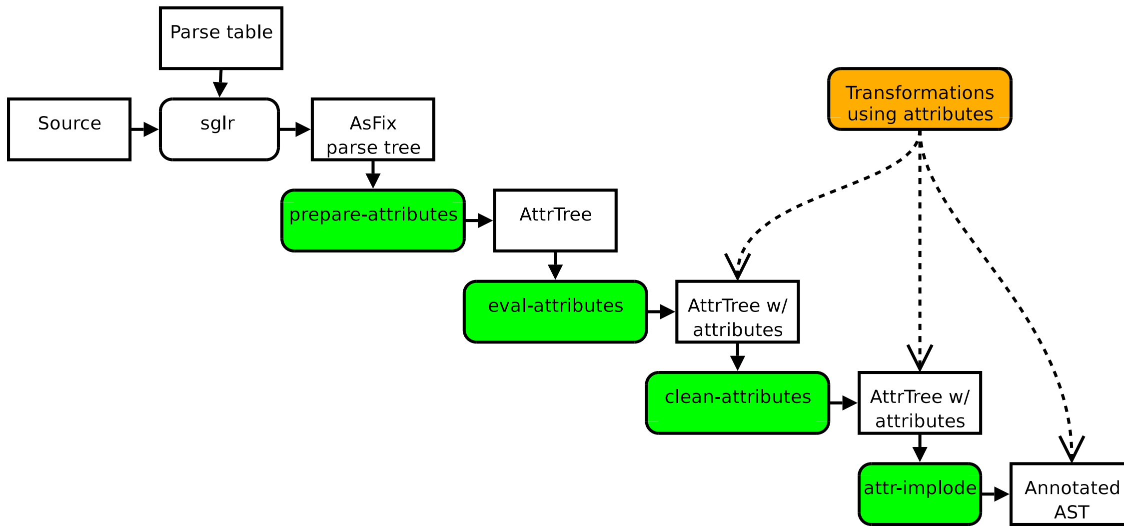

Figure 3: Evaluation process

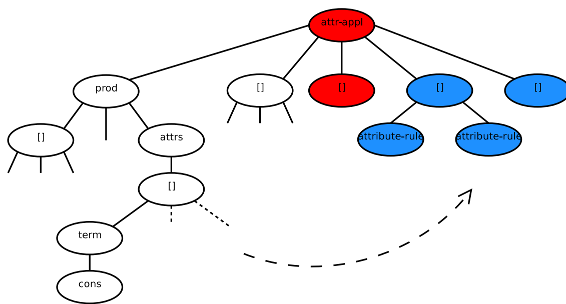

Figure 4: Making attribute more accessible

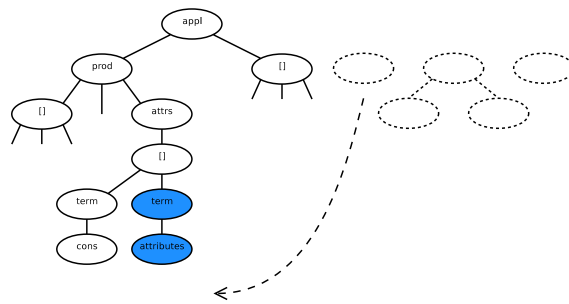

Figure 5: Back to regular AsFix

is run using our tailored parse table to parse a source file. The resulting AsFix tree has attribute rules dependency and label tables as production annotations. To ease the evaluation, `prepare-attributes` moves them a more accessible place as shown on Fig. 4. In addition, an empty list is added to accept future attribute values.

Then `eval-attributes`, the evaluator compiled from the attribute rules, evaluates the tree using dynamic rules. The traversal depends on what is computed: when new attributes are evaluated, its dependencies can be visited. Afterward, `clean-attributes` can transform the tree back into regular AsFix as shown on the figure 5. The attribute *values* are put into production rule annotations. This tree can be imploded into an AST with attribute values as annotations (`attr-implode`).

3 Discussion

This implementation of attribute grammars in SDF was developped over a few weeks, in order to provide a disambiguate-by-AG framework for the Transformers project (David et al., 2005). Many inspiring AG system exist; eventually our system will be completely rewritten to address its shortcomings.

3.1 Related Work

Of the many existing AG systems, a few caught our attention.

JastAdd II (Hedin and Magnusson, 2001) is a language implementation tool supporting generation of compilers using extended AGs: Rewritable Reference Attribute Grammars (ReRAGs) and ordinary Java code. To test their

work they implemented a Java 1.4 compiler which is only four times slower than the hand written javac compiler. They seem to be using their AG to specialize their ASTs, a (weak) form of disambiguation. Their AG system is powerful and offers interesting features to shorten the AGs.

Similarly, the UU-AG system (Baars et al., 1999), developped at the University of Utrecht, features nice concepts to factor rules. It also benefits from features of its target language, Haskell, to spare traversals.

3.2 Further Work

The current implementation of the attribute evaluator was quickly written and had for only goal to serve our needs. Hence, its implementation is naive and was designed to be as easy to implement in Stratego as it could be. As a consequence, the performance are poor, although a thorough comparison with other system was not done.

A new evaluator will be written to speed up the evaluation. The current evaluator is fully dynamic, although a lot of work can be done statically: to control the data flow between attributes an order of evaluation can be computed from specificities of the grammar and the attributes associated with.

The implementation language, and the language into which attribute rules are written is also subject to debate within our group. Some members believe that if Stratego is powerful in term rewriting, the evaluator does not need this specific feature and needs high speed in other ones.

4 Conclusion

In this paper we presented a simple but effective implementation of AGs for possibly ambiguous grammars in the world of SDF, using Stratego/XT as an implementation and execution framework. Lots of issues remain to be addressed: syntax improvement, additional features, formalization, and comparison with other information gathering schemes. This proposal nevertheless suffice to fully disambiguate ISO-C99, and even the most complex parts of C++. We are now looking forward meeting interesting in our system, and its development.

References

Baars, A., Swierstra, D., and Löh, A. (1999). UU-AG System. `http://catamaran.labs.cs.uu.nl/twiki/st/bin/view/Center/AttributeGrammarSystem`.

David, V., Demaille, A., Durlin, R., and Gournet, O. (2005). C/C++ Disambiguation Using Attribute Grammars. Submitted to Stratego Users Day 2005.

Demaille, A., Largillier, T., and Pouillard, N. (2005). ESDF: A proposal for a more flexible SDF handling. Submitted to Stratego Users Day 2005.

Hedin, G. and Magnusson, E. (2001). JastAdd. `http://www.cs.lth.se/Research/ProgEnv/rags/`.

Knuth, D. E. (1968). Semantics of context-free languages. *Journal of Mathematical System Theory*, pages 127–145. Not read, but according to all other references, it is the first text on attribute grammars.