Automatic Attribute Propagation for Modular Attribute Grammars

SVN revision: r2938

Akim Demaille, Renaud Durlin, Nicolas Pierron, Benoît Sigoure

EPITA Research and Development Laboratory (LRDE)
14-16, rue Voltaire - FR-94276 Le Kremlin-Bicêtre Cedex - France
first-name.last-name@*lrde.epita.fr* `http://www.lrde.epita.fr`

Abstract

Attribute grammars are well suited to describe (parts of) the semantics of programming languages: hooked on the syntactic production rules, they allow the expression of local relations that are later bound globally by a generic evaluator. However they fall short when working on large and complex languages. First attributes that are virtually needed everywhere have to be propagated by every single production rule. Second, this constraint hinders modularity, since adding a syntactic extension might require the propagation of new attributes in the rest of the grammar. This paper shows how to solve these problems by introducing a technique to automatically propagate attributes by completing the set of semantic rules. We formally define the propagation constraints such that it is optimized to avoid unnecessary addition of semantic rules. Attribute grammars are thus made more maintainable, modular and easier to define.

Key words: attribute grammar, context-free grammar, disambiguation, C, C++, attribute propagator

1 Introduction

Building a framework for program transformation is a tough task if the target language has not been designed for that purpose. Some languages are difficult to parse but are extremely widespread nevertheless. Frequently, needs for language extensions, static analysis and automatic refactoring arise. These needs cannot be satisfied without appropriate front-ends that strictly conform to the language specifications.

C++ [9] is one such language. It offers several powerful paradigms that

are complex to implement and use, such as the overly `templated` static algorithms [4]. A common solution is to use syntax extensions to express these paradigms with higher level constructs and provide a transformation program that rewrites the extended code in a strictly standard conforming program [1]. It is thus of utmost importance to have an extensible front-end.

TRANSFORMERS [5] provides a framework to manipulate C and C++. In order to guarantee that our implementation is strictly conforming, we chose to use the grammar exactly as specified in the standards. These grammars are highly ambiguous, especially in the case of C++. They also lead to enormous Abstract Syntax Trees (ASTs) because their grammars encode various information such as precedence rules, which can be handled more ingeniously with today's parsing techniques.

Contributions The contributions of this paper are: *(1)* A formalization of attribute propagation. *(2)* An algorithm to automatically propagate attributes in Attribute Grammars (AGs). *(3)* An implementation in TRANSFORMERS, based on STRATEGO/XT and Syntax Definition Formalism (SDF).

Outline Section 2 formalizes attribute propagation and how to automatically propagate attributes. Section 3 presents the AG compiler of TRANSFORMERS, and how well it performs on the C and C++ grammars. Section 4 analyzes related works and offers an overview of future works. Finally, Section 5 concludes.

2 Attribute Grammars

This section introduces AGs with a notation close to that introduced by Knuth [11]. We formalize attribute propagation and present an algorithm to automatically propagate Top-Down (TD), Bottom-Up (BU) and Left-to-Right (LR) attributes by generating new semantic rules.

2.1 Definitions

In Knuth [11, Section 2], AGs are introduced to specify the semantics of derivation trees. The definition introduces inherited attributes which are based on attributes of the ancestors of a symbol, as opposed to synthesized attributes, which are based on the attributes of the descendants of a non-terminal symbol.

Based on these definitions, we define the following.

- $\mathcal{V}$: the finite vocabulary of the context-free grammar (terminal and non-terminal symbols).
- $\mathcal{P}$: the set of production rules.

- Prod(X): the set of production rules that define the non-terminal symbol X (e.g, $\{X ::= Y\ Z\} \in \text{Prod}(X)$ in BNF notation).
- Use(X): the set of production rules that use the symbol X (e.g, $\{S ::= X\ Y\} \in \text{Use}(X)$ in BNF notation).
- SR(p): the set of semantic rules on the production $p \in \mathcal{P}$.
- A production $p \in \mathcal{P}$ is of the form $X_0 \rightarrow X_1 X_2 \dots X_{n_p}$ where X_0 is a non-terminal. We note p_0 to be X_0 and $p_i, i \in [\![1..n_p]\!]$ to be X_i.
- Let $X \in \mathcal{V}$ and a be an attribute. If a is associated to X, we note it $X.a$.
- Attr(X): the set of attributes (of the form $X.a$) associated to the symbol X.
- AttrNames(E): the set of attribute names a extracted from the set E of elements of the form $X.a$.

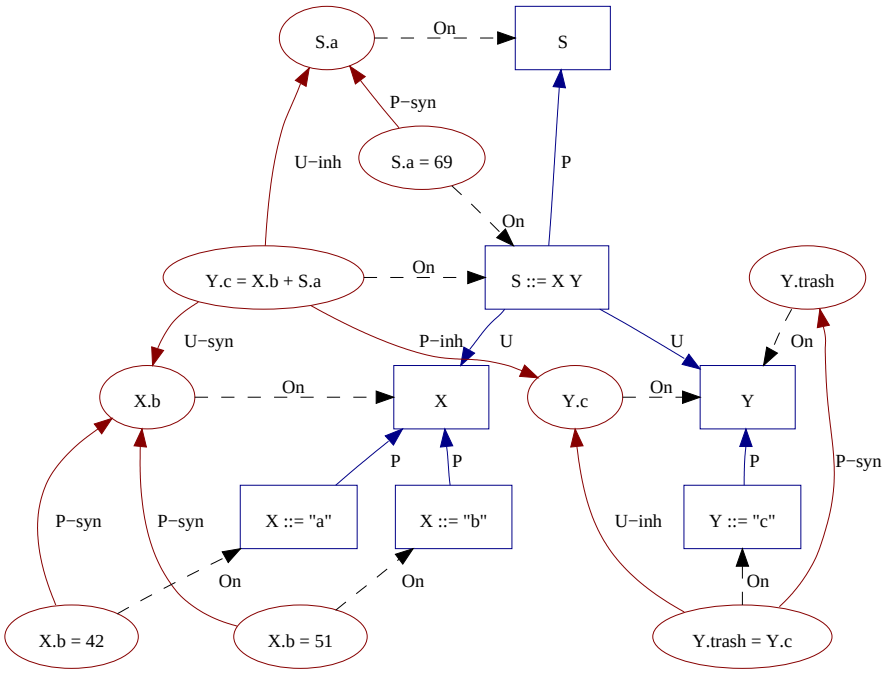
In our system, the propagation is automatic when an attribute a is used on a symbol X and produced on another Y. The bare name a is thus used to propagate a value from $Y.a$ to $X.a$. Of course, there can be and there usually is a certain number of symbols that intervene between X and Y. For instance, consider the following trivial grammar: $S ::= X$, $X ::= Y$, $Y ::= Z \dots$ where S is the start symbol and Z is a non-terminal. If the first production uses $X.a$ and the last one produces $Z.a$, the propagation will "create" $Y.a$ to transfer the value of a bottom-up. It is useful to refer both to the attributes associated to a symbol (e.g, $\text{Attr}(X) = \{X.a\}$) and to the name of the attributes in a set (e.g, $\text{AttrNames}(\text{P}_{\text{syn}}(Y ::= Z)) = \{a\}$).

2.2 Propagation properties

Attribute propagation is needed to reduce the number of semantic rules to write. In order to automatically propagate an attribute, we must know how it has to be propagated. Since our system is primarily guided by the design and implementation of a C++ front-end, we implemented three traditional propagation patterns: Top-Down (e.g, for template parameters), Bottom-Up (e.g, for type deduction) and Left-to-Right (e.g, for symbol tables). Should the need arise, it is possible to add more propagation patterns, however.

We define the following sets:

- L(p): The set of local attributes on the production p. An attribute is said to be *local* if it is only produced in a synthesized way and optionally used in an inherited way on the same production. In other words, no other production references its value, it is "scoped" in p.
- $\text{P}_{\text{syn}}(p)$ (resp. $\text{P}_{\text{inh}}(p)$): the set of non-local attributes produced by at least one semantic rule $s \in \text{SR}(p)$ in a synthesized (resp. inherited) manner.
- $\text{U}_{\text{syn}}(p)$ (resp. $\text{U}_{\text{inh}}(p)$): the set of non-local attributes used by at least one semantic rule $s \in \text{SR}(p)$ in a synthesized (resp. inherited) manner.



On this example, the following equalities are verified:

$$\mathrm{Attr}(S) = \{S.a\} \qquad \mathrm{Attr}(X) = \{X.b\} \qquad \mathrm{Attr}(Y) = \{Y.c, Y.trash\}$$

$$\mathrm{Use}(X) = \{S ::= X\ Y\} \qquad \mathrm{Prod}(X) = \{X ::= "a", X ::= "b"\}$$

$$\mathrm{SR}(S ::= X\ Y) = \{Y.c = X.b + S.a, S.a = 69\} \qquad \mathrm{AttrNames}(Y) = \{c, trash\}$$

$$\mathrm{P}_{\mathrm{syn}}(S ::= X\ Y) = \emptyset \qquad \mathrm{U}_{\mathrm{syn}}(S ::= X\ Y) = \{X.b\}$$

$$\mathrm{P}_{\mathrm{inh}}(S ::= X\ Y) = \{Y.c\} \qquad \mathrm{U}_{\mathrm{inh}}(S ::= X\ Y) = \emptyset$$

$$\mathrm{L}(S ::= X\ Y) = \{S.a\} \qquad \mathrm{L}(X ::= "c") = \{Y.trash\}$$

Figure 1: An example AG

- $\overline{\mathrm{U}_{\mathrm{syn}}}(p)$ (resp. $\overline{\mathrm{P}_{\mathrm{syn}}}(p)$, $\overline{\mathrm{U}_{\mathrm{inh}}}(p)$, $\overline{\mathrm{P}_{\mathrm{inh}}}(p)$): represents the same set as $\mathrm{U}_{\mathrm{syn}}(p)$ (resp. $\mathrm{P}_{\mathrm{syn}}(p)$, $\mathrm{U}_{\mathrm{inh}}(p)$, $\mathrm{P}_{\mathrm{inh}}(p)$) after attribute propagation.
- To simplify the formulas, we also use the following notation: $\mathrm{P}_{\mathrm{inh}}^{\mathrm{Attr}(X)}(p) = \mathrm{P}_{\mathrm{inh}}(p) \cap \mathrm{Attr}(X)$.

These sets are lifted additively to sets. In other words, we have:

$$\overline{\mathrm{P}_{\mathrm{syn}}^{\mathrm{Attr}(X)}}(\mathrm{Prod}(X)) = \bigcup_{p \in \mathrm{Prod}(X)} \overline{\mathrm{P}_{\mathrm{syn}}^{\mathrm{Attr}(X)}}(p)$$

If an attribute is used in a synthesized way, it must be produced in a

synthesized way.

$$\overline{\mathrm{U}_{\mathrm{syn}}^{\mathrm{Attr}(X)}}(\mathrm{Use}(X)) \subseteq \overline{\mathrm{P}_{\mathrm{syn}}^{\mathrm{Attr}(X)}}(\mathrm{Prod}(X)) \tag{1}$$

If an attribute is used in an inherited way, it must be produced in an inherited way.

$$\overline{\mathrm{U}_{\mathrm{inh}}^{\mathrm{Attr}(X)}}(\mathrm{Prod}(X)) \subseteq \overline{\mathrm{P}_{\mathrm{inh}}^{\mathrm{Attr}(X)}}(\mathrm{Use}(X)) \tag{2}$$

If an attribute of X is produced in a synthesized way in one production, it must be produced in a synthesized way in all productions that produce X.

$$\overline{\mathrm{P}_{\mathrm{syn}}^{\mathrm{Attr}(X)}}(\mathrm{Prod}(X)) = \bigcap_{p \in \mathrm{Prod}(X)} \overline{\mathrm{P}_{\mathrm{syn}}^{\mathrm{Attr}(X)}}(p) \tag{3}$$

If an attribute of X is produced in an inherited way in one production, it must be produced in an inherited way in all productions that use X.

$$\overline{\mathrm{P}_{\mathrm{inh}}^{\mathrm{Attr}(X)}}(\mathrm{Use}(X)) = \bigcap_{p \in \mathrm{Use}(X)} \overline{\mathrm{P}_{\mathrm{inh}}^{\mathrm{Attr}(X)}}(p) \tag{4}$$

These equations represent a well-defined AG. As opposed to the definition of Knuth, "well-defined" here does not mean "without circularity". Equations (1) and (2) mean that all attributes that are used in some way must be produced in the same way. This also ensures that an attribute cannot be used without being produced. Equations (3) and (4) mean that all attributes that are produced for one symbol must be produced on each related production of that symbol.

2.3 Propagation algorithms

Given a valid AG, the previous equations must be satisfied by the propagation. To do so, the attribute propagation algorithm must add the required semantic rules. We want to add the least number of rules, however. Indeed, we could propagate all the attributes throughout the entire grammar to satisfy the equations, but that solution is far from optimal. We define three algorithms to implement the traditional Top-Down (TD), Bottom-Up (BU) and Left-to-Right (LR) propagations. Each algorithm must find the shortest path between the "use" site and the "production" site, by following a given pattern. To do so, instead of going from the production sites to the use sites, we do the search the other way around, that is, from the use sites to the production sites. As a consequence, the propagation path for BU is discovered top-down (and *vis versa*).

For this section a new operator "$\cup=$" is introduced to express the augmentation of a set, for example $\mathrm{SR}(p) \cup= f$ is equivalent to $\mathrm{SR}'(p) = f \cup \mathrm{SR}(p)$

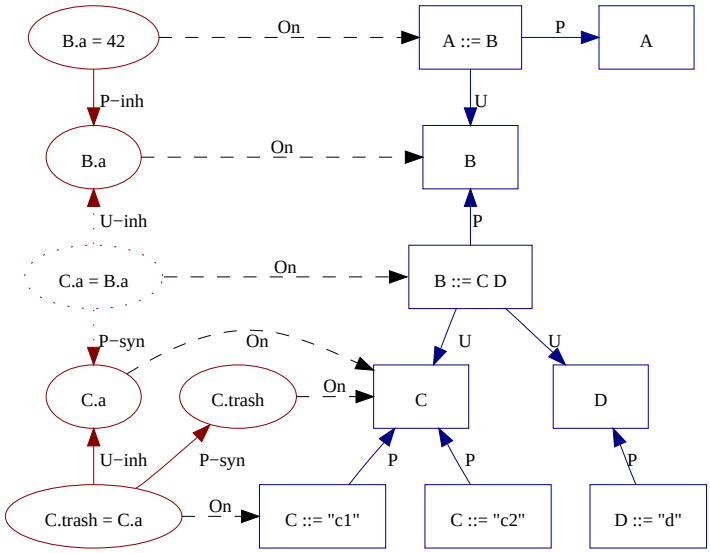


Figure 2: An example AG for TD propagation

where SR′(p) represents the extended set.

2.3.1 *Top-Down*

The TD propagation is defined as follows. Let TD be the set of all attributes propagated top-down.

For all $p \in \mathcal{P}$, $i \in [\![1..n_p]\!]$ and $a \in$ TD:

$$\frac{p_i.a \in \left(\mathrm{U}_{\mathrm{inh}}^{\mathrm{Attr}(p_i)}(\mathrm{Prod}(p_i)) \cup \mathrm{P}_{\mathrm{inh}}^{\mathrm{Attr}(p_i)}(\mathrm{Use}(p_i))\right) \setminus \mathrm{P}_{\mathrm{inh}}^{\mathrm{Attr}(p_i)}(p)}{\mathrm{Attr}(p_0) \cup\!= \{p_0.a\} \quad \mathrm{SR}(p) \cup\!= \{p_i.a = p_0.a\}}$$

In other words, for a production p that has a symbol p_i, itself involved in another production that uses or produces a in an inherited way, if p does not produce a in an inherited way, then we augment SR(p) with a rule that propagates a from p_0 to p_i. As a consequence, a also becomes an attribute of p_0 if it was not already in Attr(p_0).

2.3.2 *Bottom-Up*

For the BU propagation, we need to find a path from the "use site" to the "production site(s)" located deeper in the grammar. This problem does not occur in TD because the path from the "use site" to the "production site" (higher up in the grammar) is unique and can be trivially found by traversing the only ancestor of each production. Since a production has one ancestor but (potentially) multiple descendants, the algorithm is not trivial the other way around.

Since algorithms on AGs are easier to implement when they only make local decisions by looking at a single production, we introduce a concept of

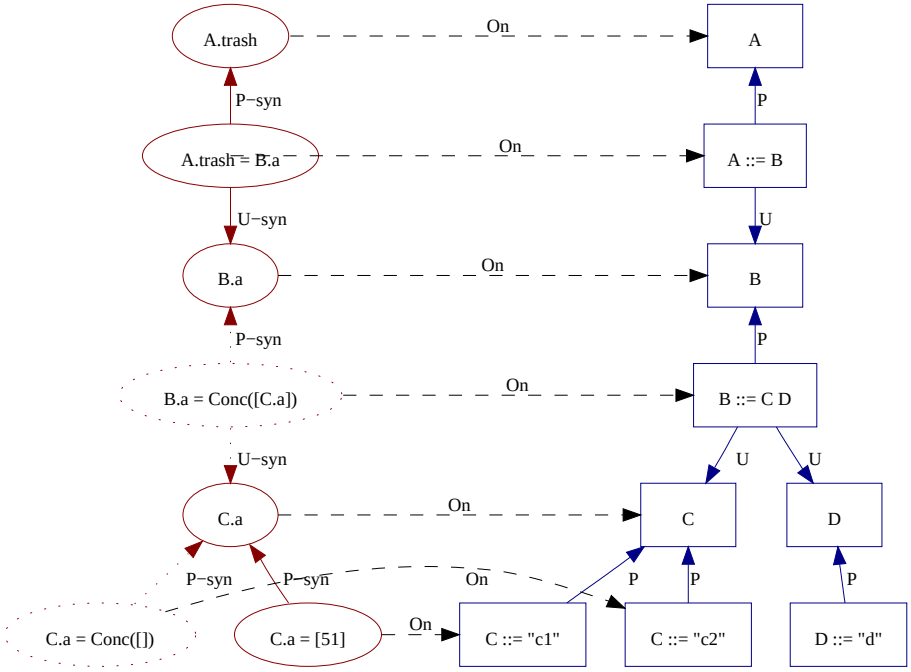


Figure 3: An example AG for BU propagation

flag to guide our BU propagation. This concept will turn out to be useful to define the LR propagation.

Before starting the process of propagation, we first assign flags to all productions in the grammar with the following formula.

For all $p \in \mathcal{P}$:

$$\text{BUFlags}(p_0) = \left(\text{AttrNames}\left(\text{P}_{\text{syn}}(p) \cup \text{L}(p)\right) \cup \bigcup_{i \geq 1} \text{BUFlags}(p_i)\right) \cap \text{BU}$$

Note that we also consider the local attributes on p. Indeed, the locality of an attribute can be caused by the fact that it is used by a production higher up in the grammar. The propagation can add a semantic rule that uses this attribute in an inherited way, making it non-local.

The BU propagation is then defined with the following equation (where BU is the set of all attributes propagated bottom-up).

For all $p \in \mathcal{P}$ and $a \in \text{BU}$ with $i \in [\![1..n_p]\!]$:

$$\frac{p_0.a \in \left(\text{U}_{\text{syn}}^{\text{Attr}(p_0)}(\text{Use}(p_0)) \cup \text{P}_{\text{syn}}^{\text{Attr}(p_0)}(\text{Prod}(p_0))\right) \setminus \text{P}_{\text{syn}}^{\text{Attr}(p_0)}(p)}{\forall i \mid a \in \text{BUFlags}(p_i) \quad \text{Attr}(p_i) \cup\!= \{p_i.a\} \quad \text{SR}(p) \cup\!= \{p_0.a = \text{Conc}(\{p_i.a\})\}}$$

Conc is a user-defined function that takes all the possible values of a coming from the various candidates p_i and merges them together. Traditionally this is implemented by producing a list that is the concatenations of all the $p_i.a$.

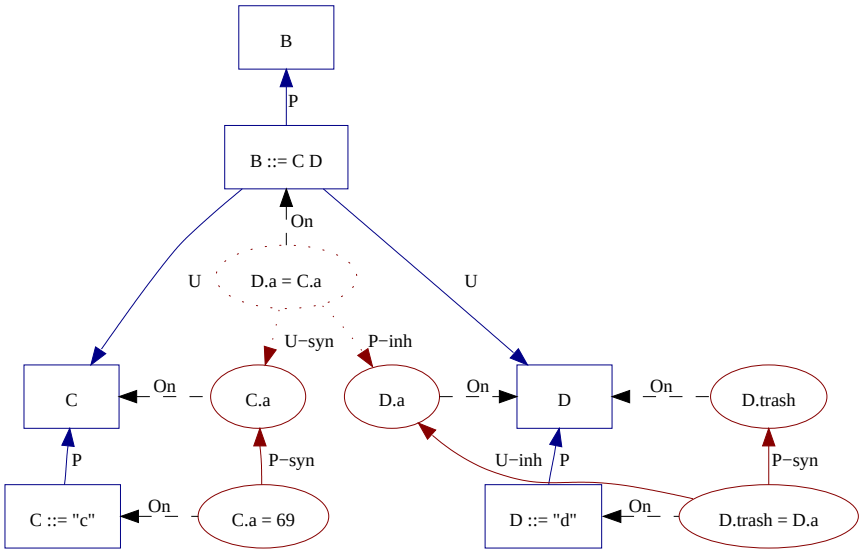


Figure 4: An example AG for LR propagation

This equation means that given a production p that defines the non-terminal p_0, if another production of p_0 produces a in a synthesized way or if another production that involves p_0 uses a in a synthesized way, and if p itself does not produce a in a synthesized way, then we augment SR(p) so that a is propagated from all the descendants p_i bearing the BU flag for a to p_0.

2.3.3 *Left-to-Right*

The implementation of LR is comparable to that of BU because we must also find a path from the "use site" to the "production site(s)". We thus tag the productions with another set of flags specific to BU propagation. Again, this needs to be done only once before any propagation begins.

For all $p \in \mathcal{P}$:

$$\mathrm{LRFlags}(p_0) = \left(\mathrm{AttrNames}\left(\mathrm{P_{inh}}(p) \cup \mathrm{P_{syn}}(p) \cup \mathrm{L}(p)\right) \cup \bigcup_{i \geq 1} \mathrm{LRFlags}(p_i) \right) \cap \mathrm{LR}$$

In order to define the LR propagation, we need the following auxiliary function that finds the first descendant of a production that bears the LR flag for a given attribute.

$$\underset{p \in \mathcal{P}}{\mathrm{rightmost}}(p, a, n) = \begin{cases} 0 & \text{if } n = 0 \\ n & \text{if } a \in \mathrm{LRFlags}(p_n) \\ \mathrm{rightmost}(p, a, n-1) & \text{otherwise} \end{cases}_{a \in \mathrm{LR} \quad n \in [\![0..n_p]\!]}$$

The LR propagation is defined with the following equations (where LR

is the set of all attributes propagated left-right).

For all $p \in \mathcal{P}$ and $a \in \mathrm{LR}$:

$$\frac{p_0.a \in \left(\mathrm{U}_{\mathrm{syn}}^{\mathrm{Attr}(p_0)}(\mathrm{Use}(p_0)) \cup \mathrm{P}_{\mathrm{syn}}^{\mathrm{Attr}(p_0)}(\mathrm{Prod}(p_0))\right) \setminus \mathrm{P}_{\mathrm{syn}}^{\mathrm{Attr}(p_0)}(p)}{j = \mathrm{rightmost}(p, a, n_p) \quad \mathrm{Attr}(p_j) \cup = \{p_j.a\} \quad \mathrm{SR}(p) \cup = \{p_0.a = p_j.a\}}$$

This equation means that given a production p that defines the non-terminal p_0, if another production of p_0 produces a in a synthesized way or if another production that involves p_0 uses a in a synthesized way, and if p itself does not produce a in a synthesized way, then we augment $\mathrm{SR}(p)$ so that a is propagated from the rightmost non-terminal which produces a to p_0. If no descendants of p bear the expected attribute then an ancestor production of p should bear the attribute.

For all $p \in \mathcal{P}$, $i \in [\![1..n_p]\!]$ and $a \in \mathrm{TD}$:

$$\frac{p_i.a \in \left(\mathrm{U}_{\mathrm{inh}}^{\mathrm{Attr}(p_i)}(\mathrm{Prod}(p_i)) \cup \mathrm{P}_{\mathrm{inh}}^{\mathrm{Attr}(p_i)}(\mathrm{Use}(p_i))\right) \setminus \mathrm{P}_{\mathrm{inh}}^{\mathrm{Attr}(p_i)}(p)}{j = \mathrm{rightmost}(p, a, i-1) \quad \mathrm{Attr}(p_j) \cup = \{p_j.a\} \quad \mathrm{SR}(p) \cup = \{p_i.a = p_j.a\}}$$

This equation means that given a production p that has a symbol p_i, itself involved in another production that uses or produces a in an inherited way, if p does not produce a in an inherited way, then we augment $\mathrm{SR}(p)$ with the same semantic rule as the previous equation except that the right most descendant of p can only be searched at the left-side of the non-terminal p_i.

These algorithms always check whether a semantic rule is missing to satisfy the properties enunciated in Section 2.2 before trying to add one. Therefore, an attribute is produced at most once per grammar production. In addition since the algorithms manipulate growing sets, the same attribute cannot be added twice in a given set. Moreover these sets of attributes are majored by the set that contains all attributes of the grammar. Consequently these equations, which do not remove either attributes or semantic rules and do not modify the grammar, can be used until a fixed-point is reached, which satisfies the equations of a well-defined AG.

3 Transformers Attribute Grammars

This section presents our system which uses AGs to disambiguate the C and the C++. First it introduces the framework used by Transformers. Then it delves deeper in the implementation techniques we used.

3.1 *Framework*

The framework of Transformers is based on Stratego/XT [3]. Its aim is to provide a generic, extensible, strictly standard conforming C++ front-end. By using Stratego, we reap the benefits of Syntax Definition Formalism (SDF) [14] which comes with a generic Scanner-less Generalized LR (SGLR) parser [15] and pretty printer [10].

Two primary reasons motivate the design of a C/C++ front-end in SDF. Firstly, SDF fosters modularity and extensibility. Secondly, this is mandated to easily use C/C++ concrete syntax in Stratego [1]. With our approach, anyone willing to extend C++ can do so by adding SDF modules and extending the AG. No modification of the existing C++ front-end is required whatsoever.

The advantage of SGLR is that ambiguities are preserved. Our tools, based on Stratego/XT, use ATerm as input and manipulate the AST or Parse Tree (PT) to disambiguate. Memory-wise, ATerm is advantageous thanks to its maximal sharing. It efficiently represents the large ambiguous parse forests of C++.

In order to express AGs directly in SDF, with annotations on the productions, we extended SDF. We designed another extension to handle the pretty-print table similarly. Extending SDF is easy thanks to the Stratego/XT framework. Our tools desugar Extended SDF (ESDF) files down to standard SDF modules.

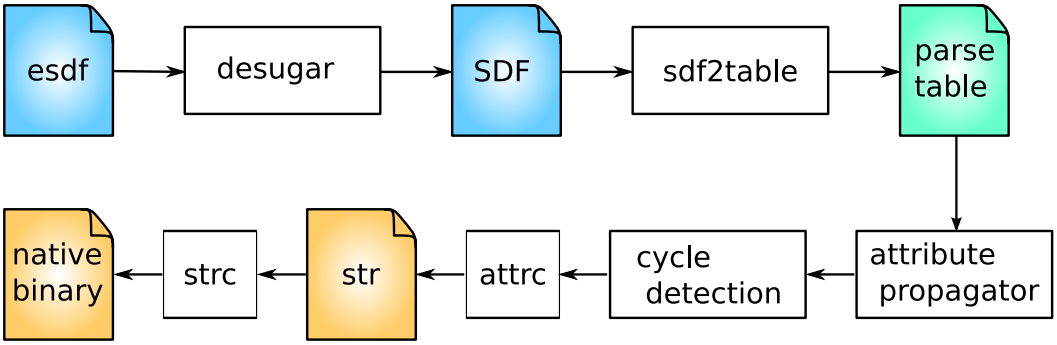
The advantage of putting everything in SDF modules is that we tie the relevant information together, instead of spreading it in multiple independent files (for the grammar, pretty-print rules, AG ...) that must remain synchronized. Thanks to the modularity of SDF, one can design a front-end that isolates the different parts of the grammar in different modules. This is an extremely valuable asset if you want to reuse parts of a grammar (such as the C++ `constant-expression` used to implement a C++ pre-processor, for instance) or to assimilate a part of a language in another [2]. Our experience shows that the resulting front-end is much more maintainable.

The semantic rules are written in Stratego so as to directly manipulate ASTs and PTs with ATerm and to benefit from the well-furnished standard libraries of Stratego/XT.

3.2 *Compilation process*

Our AG compiler is divided into five steps, the transformation to a core language, extraction of the various information from the ESDF, attribute propagation, cycle detection and generation of an evaluator.

The compilation pipeline depicted on Figure 5 consists mainly in extracting the various information from the ESDF modules (SDF grammar,



The ESDF is desugared down to SDF, from which a parse table is generated. This parse table contains the AG on which we propagate attributes. Then we ensure the propagated AG is cycle-free before compiling it with `attrc` which generates an evaluator in Stratego.

Figure 5: Compilation Pipeline

Language	AG	→	Stratego	→	C	→ Binary	Total
C	3.3KL	21s	9.5KL	61s	102KL	63s	145s
C++	6.5KL	70s	28KL	190s	237KL	195s	455s

Timings performed on an AMD Athlon XP 3000+ (2.1 Ghz) with 512M of DDR RAM, running GNU/Debian Linux with the kernel 2.6. We use the Stratego compiler version 0.17 and GCC 4.1 for the C back-end.

Table 6: Number of lines (KL: Kilo-lines) & Compilation time (seconds).

Language	User-defined rules	Generated rules	Total
C	166 (27.0%)	448 (72.9%)	614
C++	995 (37.3%)	1671 (62.6%)	2666

Table 7: Ratio Explicit (User) / Implicit (Added) semantic rules.

pretty-print rules, AG, ...) and then pass them to the standard tools of Stratego/XT. Our tools extend the Stratego/XT toolchain in various places. We provide `Makefiles` to ease the compilation process.

Our attribute grammar compiler uses the cycle detection algorithm described by [12, Section 6]. If the grammar is well-defined, it is compiled to a Stratego evaluator, itself compiled in a native binary by the Stratego/XT toolchain (which uses C as an intermediate language). The compilation times are depicted on Table 6.

Table 7 confirms that most of the semantic rules can be generated by an automatic propagator. Our compiler also implements an optimizations for aliases. Indeed, when a semantic rule is only transfering a value from a

Language	Aliased	Non-Aliased	Total
C	469 (76.3%)	145 (23.6%)	614
C++	1920 (72.0%)	746 (27.9%)	2666

Table 8: Number of semantic rules that are replaced by Aliases.

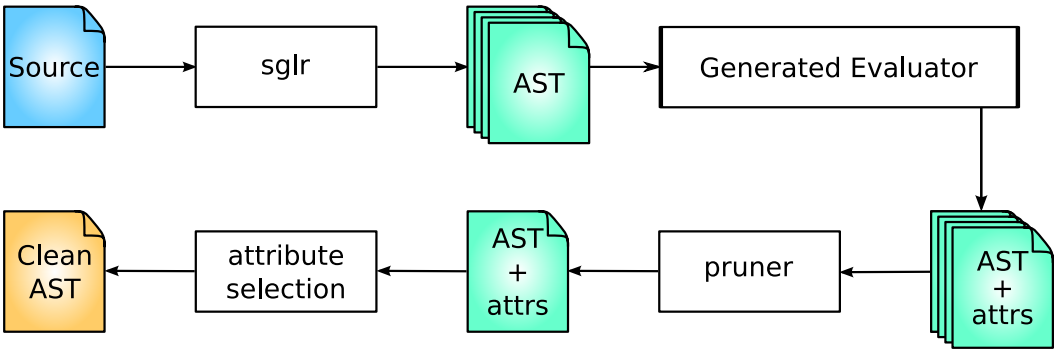


Figure 9: Execution Process

symbol to another, it can be optimized out with an alias. Aliasing considerably optimizes the runtime, especially in terms of memory usage, since an important number of semantic rules can be optimized, as shown in Table 8.

3.3 Execution

The execution is divided into four steps: parsing, evaluation, pruning and attribute selection. The process is depicted on Figure 9.

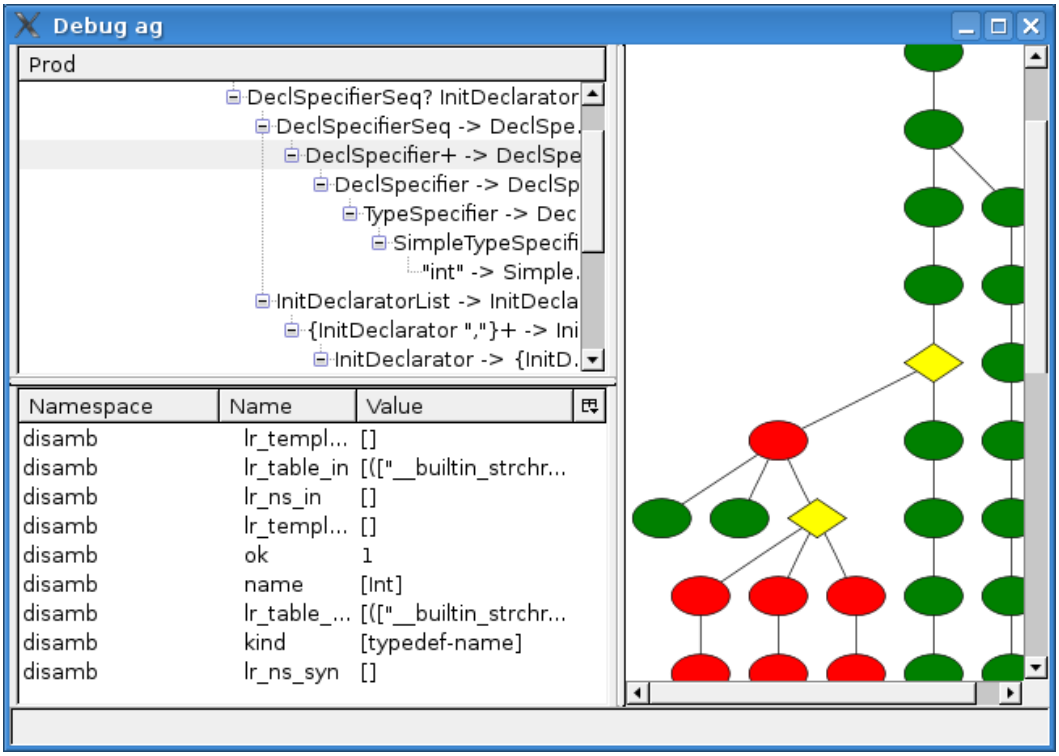
SGLR generates the parse-forest of all the possible derivation trees which our generated evaluator attributes. Then the attributed parse-forest of ASTs is evaluated and invalid branches are pruned to eventually yield the only valid derivation tree. The attributes that are not required by the subsequent transformations or analysis are removed from the AST.

Our implementation reports errors in the input source (e.g, multiple derivation trees are still valid because the code is ambiguous) to the user, but the error reporting is not as sound as that found in today's C and C++ compilers.

3.4 Development tools

AG systems are difficult to debug when the error must be found in large ATerm outputs. We implemented some debugging tools to help the user visualize the propagation.

Large PTs formated in ATerm are really difficult to read and are even less readable with attributes attached on each non-terminal symbol. For a long time this has been a real struggle in the development of Transformers.



The main window presents the tree on which remaining ambiguous nodes are displayed with yellow diamonds and each other node has a color to summarize the status of its attributes. The top left window is used to reference the production rules involved in this particular derivation tree. The bottom left window gives access to attribute values. All views are synchronized to simplify the navigation in the tree.

Figure 10: Debugger

We implemented a graphical debugging tool that filters attributes by their name or namespace, and convert the resulting PT or AST to a dot-file [7] which can then be converted to a SVG file. Our graphical user interface reads the SVG file and presents the derivation tree with colors and details about each attribute and each symbol as depicted on Figure 10.

One advantage of the attribute propagator is its ability to generate code. Our propagator can issue warnings when the user writes semantic rules that could be generated automatically. This helps the user keep the minimal number of semantic rules in their ESDF modules.

4 Discussions

4.1 Related work

UU-AG [13] is an AG system in Haskell [8] where semantic rules are written in Haskell. Each attribute has to be declared on every symbol of the grammar where it is used. Attributes are declared inherited or synthesized or both to define default propagations. Each synthesized attribute can be declared with a default Conc function and an initial value if the set of attribute is empty. Each inherited and synthesized attribute is propagated as described in Section 2.3.3. Thus UU-AG seems to be the closest system except that this approach is not really suitable for disambiguation problems because of the declaration of a generic ambiguous node. In addition attribute declarations lead the system to a non-optimized attribute propagation in the case of a user error (e.g, if one declares that attributes a is to be propagated from symbol X to symbol Y whereas the semantic rules do not actually require this, the value of a will be propagated anyway). Furthermore, the wildcard notation, which propagates an attribute on all symbols, forbids modularity (e.g, it is impossible to re-use a part of the AG where such attribute is not defined).

JastAdd [6] uses different technologies and thus a different approach. They rely on Java and object-oriented programming to express the AG and manipulate it. In JastAdd, synthesized attributes are virtual methods without side-effects. This helps to reduce the amount of code to propagate values across large grammars. Inherited attributes are propagated to all the descendants of the node that produced it, which is sub-optimal compared to our propagation. The framework was also designed to be extensible and modular. There are several problems, however, which make it inappropriate in the context of Transformers. JastAdd relies on third-party Java parser generators. Even though implementations of generalized parsers in Java are currently being developed, the framework is not suitable for disambiguation, one of our main concerns. Moreover, it has no representation for ambiguities, so they must be encoded in the grammar (such as the infamous `PackageNameOrType` node in Java front-ends) which does not scale for grammars such as that of C++.

4.2 Future work

In our implementation, attributes start to exist as soon as they are used in a semantic rule. We want to add declaration blocks to ease maintenance and have an appropriate place to document the attributes. Additionally, we want to express invariants on attributes when they are declared. This would help constrain their domain of definition, for instance.

Currently, the TD, BU and LR propagations are hard-coded in our propagator. We want to be able to easily define new propagations schemes directly in our AG to remove several semantic rules that propagate attributes with less common patterns. The idea is to dynamically load Stratego modules that implement the propagation schemes at runtime. This would greatly facilitate the extensibility of our propagator.

To ease disambiguation, we would like to improve our system with respect to automatic branch pruning. Currently, the user has to add an extra attribute, usually named `ok`, to indicate whether a branch is valid or not. Without this attribute, branches with some invalid attributes could synthesize their valid values. The user has to manually check that no value is synthesized from a branch where `ok` is invalid. We would like to do this verification automatically. This would greatly enhance the maintainability of AGs by reducing the complexity of semantic rules in the context of disambiguation. One idea to tackle this issue is to automatically adjust the dependency graph of the evaluator, but this remains an open issue.

5 Conclusion

In this paper, we outlined several formal properties of Attribute Grammars (AGs) and defined what is a "well-defined" AG in the context of automatic propagation. We presented an iterative fix-point algorithm to implement automatic attribute propagation for three traditional patterns: Top-Down (TD), Bottom-Up (BU) and Left-to-Right (LR). The algorithm adds semantic rules to propagate the values along optimal paths throughout the AG.

Transformers, a framework with two generic front-ends to manipulate C and C++, implements this algorithm in its AG compiler. We show how automatic propagation is effective and scales to complex and ambiguous grammars such as that specified by the C++ standard [9]. Indeed, about two thirds of the semantic rules are generated by our propagator in the case of C++. AGs are thus made easier to define and maintain, by reducing the number of semantic rules written by the user. Additionally, Transformers benefits from the modularity and extensibility of the Syntax Definition Formalism (SDF) [14] and from the power of Stratego/XT [3], a framework dedicated to program transformation. It is thus possible to use our AG compiler to create front-ends that can be re-used, extended or assimilated in other front-ends.

Acknowledgment
Steve Frank for taking the time to proofread the paper.

References

[1] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

[2] M. Bravenboer and E. Visser. Designing syntax embeddings and assimilations for language libraries. In *Workshop on Language Engineering (ATEM'07)*, Nashville, USA, October 2007. (to appear).

[3] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16. Components for transformation systems. In *ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM'06)*, Charleston, South Carolina, January 2006. ACM SIGPLAN.

[4] N. Burrus, A. Duret-Lutz, Th. Géraud, D. Lesage, and R. Poss. A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming. In *Proceedings of the Workshop on Multiple Paradigm with OO Languages (MPOOL)*, Anaheim, CA, USA, Oct. 2003.

[5] V. David, A. Demaille, and O. Gournet. Attribute grammars for modular disambiguation. In *Proceedings of the IEEE 2nd International Conference on Intelligent Computer Communication and Processing (ICCP'06)*, Technical University of Cluj-Napoca, Romania, Sept. 2006.

[6] T. Ekman. A case study of separation of concerns in compiler construction using JastAdd II. In *The Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, 2004.

[7] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[8] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-X.

[9] ISO/IEC. ISO/IEC 14882:2003 (e). Programming languages — C++, 2003.

[10] M. d. Jonge. Pretty-printing for software reengineering. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 550–559, Washington, DC, USA, Oct. 2002. IEEE Computer Society. ISBN 0-7695-1819-2.

[11] D. E. Knuth. Semantics of context-free languages. *Journal of Mathematical System Theory*, pages 127–145, 1968.

[12] M. Rodeh and M. Sagiv. Finding circular attributes in attribute grammars. *Journal of the ACM*, 46(4):556–575, 1999.

[13] S. D. Swierstra, A. Baars, and A. Löh. The UU-AG attribute grammar system. `http://www.cs.uu.nl/wiki/bin/view/HUT/AttributeGrammarSystem`, 2003.

[14] E. Visser. A family of syntax definition formalisms. In M. G. J. van den Brand et al., editors, *ASF+SDF'95. A Workshop on Generating Tools from Algebraic Specifications*, pages 89–126. Technical Report P9504, Programming Research Group, University of Amsterdam, May 1995.

[15] E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.