Formal Definition of Disambiguation with Attribute Grammars

Nicolas Pierron

Technical Report n^o0702, June 2007
revision 1434

`Bibtex reference`: pierron.07.seminar.formal.def

The current problem of the disambiguation in Transformers with attribute grammars is that no-one has a proof that allows certification of this approach. The current use of attribute grammars for the disambiguation of C and a part of C++ lets us think that this method is correct.

In order to remove any doubt, a definition and a formalization of our approach are necessary. This work is split in two. The first part relates to the proof of the validity of the approach used in Transformers. The second part is devoted to the correction and the Re-development of the existing tools in order to correspond to the definite model.

Keywords
disambiguation, attribute grammars, attribute propagation, Transformers

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
nicolas.pierron@lrde.epita.fr – http://publis.lrde.epita.fr/200706-Seminar-Pierron

Copying this document

Contents

Chapter 1

Introduction

TRANSFORMERS is a project that has to make code transformations on C++ programs. To accomplish this goal we first have to parse and disambiguate the C++.

The important feature of TRANSFORMERS it is our will to keep the grammar describe in the C++ standard (ISO/IEC, 2003). The main problem with this grammar is that it has not been created to be parsed. Most implementation of C++ parser have modified the grammar in order to parse all C++ constructions with a reduce number of ambiguities.

In TRANSFORMERS we have chosen to parse any C++ program with the standard grammar and a Scanner-less Generalized LR (SGLR) parser. This parser generates a parse-tree that contains a huge number of ambiguities. To remove ambiguities we use an Attribute Grammar (AG) to disambiguate the C++.

Since 2004, TRANSFORMERS uses AGs to disambiguate. For 3 years we have tried to use AGs to disambiguate the C (ISO/IEC, 1999) and the C++. Today, in 2007, the C++ disambiguation is not functional as we have expected. After 3 years, we think it would be important to know if our system is capable to disambiguate any source code.

The purpose of this work is to verify that our system is able to disambiguate any source code similar to a C or C++ code.

This work has offer the opportunity to rebuild the attribute propagation system from the beginning. This would become the first definition of a tool to automatically propagate attributes in an AG as far as we know.

Therefore, we introduce a notation that would help for the manipulation of AG. Then we can study the problem of attribute propagation and the problem of disambiguation with AG.

Acknowledgments

Renaud Durlin for helping me by reporting some bugs of the new attribute propagator.

Stephen-Joseph FRANK and Thomas MOULARD for reading a draft of this report.

Microsoft Word 2003 for correcting a great deal of grammar mistakes.

Chapter 2

Notations

This chapter introduces Attribute Grammars (AGs) with a different notation compared to the notation introduced by Knuth (1968). First it reminds the notation of AGs. Second it describes the hybrid logic. Third it makes an equivalence relation of notation to express AGs. The reason is that the hybrid logic notation can be represented easily as a graph instead of the usual AG notation.

2.1 Attribute Grammars

This section is extracted from the article of Knuth (1968, section 2) where he introduces the first notation for attributes grammars. Some parts have been removed because they aren't required in our usage of AGs or can be satisfy easily by small transformations.

A context free grammar is defined by $\mathcal{G} = (\mathcal{V}, \mathcal{N}, S, \mathcal{P})$ where $\mathcal{V}$ is the (finite) vocabulary of terminal and nonterminal symbols; $\mathcal{N} \subseteq \mathcal{V}$ is a set of nonterminal symbols; $S \in \mathcal{N}$ is the "start symbol", which appears on the right-hand side of no production rule; and $\mathcal{P}$ is a set of production rules.

Semantic rules are added to $\mathcal{G}$ in the following manner: To each symbol $X \in \mathcal{V}$ we associate a finite set $A(X)$ of attributes; $A(X)$ is partitioned into two disjoint sets, the *synthesized attributes* $A_0(X)$ and the *inherited attributes* $A_1(X)$.

Let $\mathcal{P}$ consist of m productions, and let the pth be

$$X_{p0} \rightarrow X_{p1} X_{p2} \ldots X_{pn_p} \tag{2.1}$$

where $n_p \geq 0$, $X_{p0} \in \mathcal{N}$, and $X_{pj} \in \mathcal{V}$ for $1 \leq j \leq n_p$. The semantic rules are $f_{pj\alpha}$ defined for $1 \leq p \leq m$, $0 \leq j \leq n_p$, and $\alpha \in A_0(X_{pj})$ if $j = 0$, and $\alpha \in A_1(X_{pj})$ if $j > 0$. Each semantic rule maps values of certain attributes of $X_{p1}, X_{p2}, \ldots, X_{pn_p}$ onto the value of some attribute of X_{pj}

2.2 Hybrid Logic

This section presents the notation of the hybrid logic language (Blackburn, 2000). This language derived from the modal logic language. It provides us an explicit expression which is evaluated with a graph as model.

In this report, all expressions of hybrid logic used symbols described in the Table 2.1. Those symbols are used to define the hybrid logic language.

n	Represents a variable which could be bind to a node.
φ	Represents hybrid logic expressions.
$p(..)$	Represents a property.
$P(\varphi, ..)$	Represents a predicate.

Table 2.1: Hybrid logic symbols.

Table 2.2 defines all constructions valid in an hybrid logic expression. This table is divided in two parts: the syntax, the semantic. Hybrid logic expressions are evaluated on a graph. An expression is always evaluated on a node of the graph. In normal cases only expressions which start with one of these quantifiers are valid: $@n$ where n is a node of the graph; $\forall$; $\exists$.

$\top$	This expression is always true.
$\bot$	This expression is always false.
n	This expression is true only if the current node is the node n.
$@n\ \varphi$	This expression is true only if φ is true for the node n.
$\downarrow n\ \varphi$	This expression binds the node n to the current node and is true only if φ is true.
$\neg\varphi$	This expression is true only if φ is false.
$\varphi \vee \psi$	This expression is false only if φ and ψ are false.
$\varphi \wedge \psi$	This expression is true only if φ and ψ are true.
$\varphi \Rightarrow \psi$	This expression is true only if φ and ψ are true or if φ is false.
$\forall\ \varphi$	This expression is true only if φ is true for all nodes.
$\exists\ \varphi$	This expression is true only if φ is true for one node.
$[R]\ \varphi$	This expression is true only if all nodes reachable by relation R verify the expression φ.
$\langle R \rangle\ \varphi$	This expression is true only if one node reachable by relation R verify the expression φ.

Table 2.2: Hybrid logic language.

In the previous table the relation R can be seen as an edge in a directed graph. These relations can handle some attributes described in Table 2.3

$\dagger$	Inverse relation.
$*$	Reflexivity relation.
$+$	Transitivity relation.

Table 2.3: Relations operators.

The dagger symbol (†) is frequently used in this report. It means that you are following the edge in the opposite direction. The following expression describes that for any models if you can reach the node n_2 by following the relation R from the node n_1 then you can reach the node n_1 by following the relation R in the opposite direction from the node n_2.

$$\models @n_1 \ \langle R \rangle \, n_2 \Rightarrow @n_2 \ \left\langle R^\dagger \right\rangle n_1 \tag{2.2}$$

Be careful with the Equation 2.2, you cannot change the $\langle R \rangle$ expression by the $[R]$ expression on both sides.

In this report the following notations are strictly followed:

- All nodes are written with small caps like "NODE".
- All relations are written in italic like "*relation*" and abbreviated with the initial in all math expressions like "R".
- all relations' annotations are written as subscript of the relation in all math expressions like "$R_{annotation}$".
- All predicates are written in bold like "**Predicate**".

Any use of the relation R would refer to the relation R even if the relation is annotated in the graph. This allows us to write relation hierarchies.

2.3 Hybrid Logic for Context-Free Grammars

This section defines a bridge between the context-free grammar notation and the hybrid logic notation.

There are two kinds of nodes related to the context-free grammar definition: a terminal or a nonterminal is a SYMBOL; and a production is a PRODUCTION[1].

Those nodes are joined by two kinds of relations, as described in the Equation 2.1: In the production P, all X_{pj} where $0 < j \leq n_p$ are *used* by the PRODUCTION P; and X_{p0} is *produced* by the PRODUCTION P. So the PRODUCTION P *uses* all SYMBOLS from X_{p1} to X_{pn_p} and *produces* X_{p0}

In TRANSFORMERS we have to propagate attributes on the grammar and return the expected grammar. This step is easier to do with a graph instead of a list of productions. So to be able to come back to the previous notation we need a bijective transformation. So it is required to add the position in the production as an annotation to the *produce* relation. So all *use* relations have to keep the rank j of each SYMBOL *used* by the PRODUCTION.

The following list of expressions expresses limitations of context-free grammars:

- A PRODUCTION has to *produce* only one SYMBOL.

$$\forall \ \downarrow s \ (\text{is-symbol} \Rightarrow \left[P^\dagger\right] (\text{is-production} \Rightarrow [P] \ s))$$

[1] This production is a node

- A PRODUCTION can *use* a SYMBOL at the position n if this PRODUCTION *uses* a SYMBOL at the position $n-1$ or if $n=1$. The following expression is used to specify the maximum rank of a production in a context-free grammar.

$$\forall \text{ is-production} \Rightarrow (\langle U_n \rangle \text{ is-symbol} \Rightarrow \text{equal}(n,1) \vee \langle U_{n-1} \rangle \text{ is-symbol})$$

- A SYMBOL can only be *used* by a PRODUCTION. The following expression is used to add type check information to distinguished nodes.

$$\forall \text{ (is-symbol} \Rightarrow \left[U^{\dagger}\right] \text{ is-production)}$$

- A SYMBOL can be *produced* by at least one PRODUCTION. The following expression does the same thing as the previous one.

$$\forall \text{ (is-symbol} \Rightarrow \left\langle P^{\dagger} \right\rangle \text{ is-production)}$$

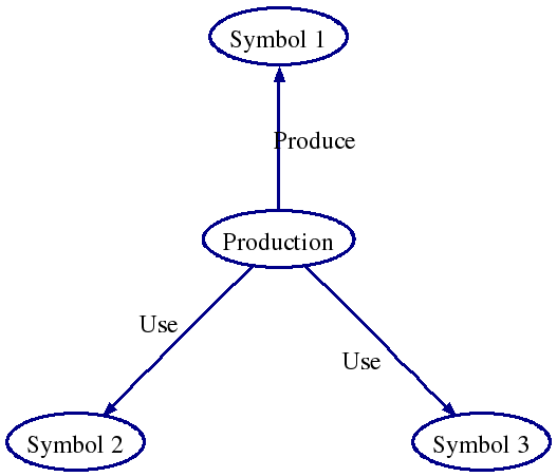

Figure 2.4: Example of a grammar model.

Figure 2.4 summarizes a model used in hybrid logic to represent a grammar in this report.

2.4 Hybrid Logic for Attribute Grammars

This section defines AGs in hybrid logic from the definition specified in the section 2.1 of this report.

In addition to previous nodes introduced in the description of the context-free grammar, there are two more nodes. One is used to represent ATTRIBUTE and the other used to represent STRATEGY. A STRATEGY is like a PRODUCTION in that it *uses* and *produces* ATTRIBUTES.

Following expressions describe type limitations of the model:

- A STRATEGY has to *produce* only one ATTRIBUTE.

$$\forall \downarrow a\ (\text{is-attribute} \Rightarrow \left[P^{\dagger}\right]\ (\text{is-strategy} \Rightarrow [P]\ a))$$

- A STRATEGY can only *use* ATTRIBUTES.

$$\forall\ \text{is-strategy} \Rightarrow ([U]\ \text{is-attribute})$$

- An ATTRIBUTE can only be *used* by a STRATEGY.

$$\forall\ (\text{is-attribute} \Rightarrow \left[U^{\dagger}\right]\ \text{is-strategy})$$

- An ATTRIBUTE can be *produced* by at least one STRATEGY.

$$\forall\ (\text{is-attribute} \Rightarrow \left\langle P^{\dagger}\right\rangle\ \text{is-strategy})$$

To link the definition of context-free grammar with semantic rules, we have to add the relation "*on*". This describes that STRATEGIES and ATTRIBUTES are respectively linked to PRODUCTIONS or SYMBOLS.

The following expressions describe how ATTRIBUTES and STRATEGIES are related to the context-free grammar:

- A STRATEGY is always *on* one PRODUCTION.

$$\forall\ \downarrow p\ (\text{is-production} \Rightarrow \left[O^{\dagger}\right]\ (\text{is-strategy} \Rightarrow [O]\ p))$$

- An ATTRIBUTE is always *on* one SYMBOL.

$$\forall\ \downarrow s\ (\text{is-symbol} \Rightarrow \left[O^{\dagger}\right]\ (\text{is-attribute} \Rightarrow [O]\ s))$$

- For all PRODUCTION "p" where *there is* a STRATEGY, all **Related** ATTRIBUTES are *on* SYMBOL **Related** to the PRODUCTION "p".

$$\text{Related}^{\dagger}(\varphi) = \left\langle P^{\dagger}\right\rangle \varphi \vee \left\langle U^{\dagger}\right\rangle \varphi$$

$$\neg\,\text{Related}(\neg\varphi) = [P]\,\varphi \wedge [U]\,\varphi$$

$$\forall\ \downarrow p\ (\text{is-production} \Rightarrow \left[O^{\dagger}\right]\ \neg\,\text{Related}(\neg\,[O]\,\text{Related}^{\dagger}(p)))$$

The propagation system is easier to write if there is a way to know if the attribute is *synthesized* or *inherited* as described in the article by Knuth (1968). It is easy to write because we have to know if the propagation direction is the same as the production in the grammar (*synthesized*) or the opposite (*inherited*). To satisfy this we have to add annotations on all *produce* and *use* relations related to an ATTRIBUTE.

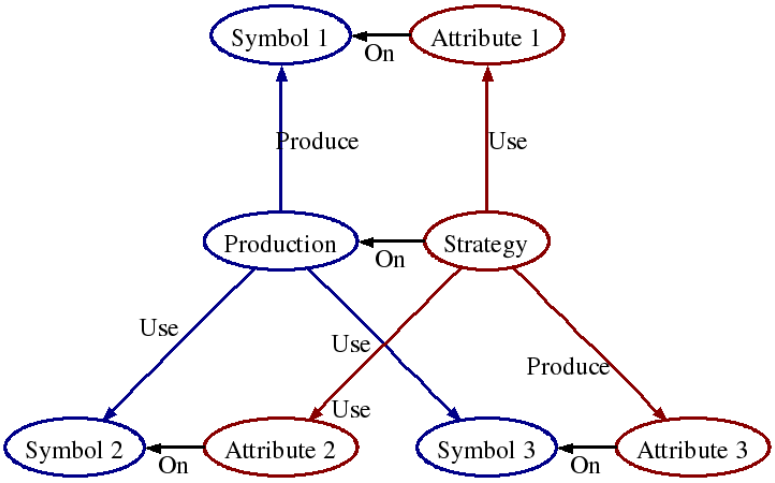

Figure 2.5: Example of an AG model.

Figure 2.5 summarizes a model used in hybrid logic to represent an AG in this report.

2.5 Implementation Similitude

This section is used to show that the implementation used for the new propagator is very similar to expressions of hybrid logic. As seen in previous sections the hybrid logic is used to express expression on a graph. The graph is the model.

In any implementation you have to use conditions to choose between processes. The hybrid logic is advantageous because graphs can be implemented in our programming language. In addition the hybrid logic language facilitates the writing of conditions on graphs.

To implement this in STRATEGO we have use a new extension of the STRATEGO compiler (Kalleberg and Visser, 2006). We have to use an extension of the compiler because the normal compiler, offer with the STRATEGO/XT framework (Bravenboer et al., 2006), does give us a support for graph manipulation. For some performance reasons we have implemented a new version of this extension. This new version is not safe with back-tracking operations. Therefore we have not to keep all states of the graph. This implies that all modifications produce a side effect if a back-tracking operation is use over the modification part. This implementation has also been created to be at least a bit compatible with the original extension.

The implementation has been made to correspond to the hybrid logic language. It defines all constructions describe in section 2.2 to fit the language. These constructions are useful to express conditions on the graph. With these constructions we are now able to verify if our conditions are correct.

Chapter 3

Attribute Propagator

The reason for this chapter is that the previous attribute propagator was not correct. So this chapter is dedicated to formalizing a method to propagate attributes in a grammar. The first section describes the common part of each propagation. The second section is more precise.

3.1 General Propagation

The attribute propagation is needed. This is the case in the C++ grammar when we can have trees with great depth. The aim of the attribute propagation is to reduce the amount of code to make a small modification.

In the approach of Swierstra et al. (2003), the user has to declare all attributes that are on each symbol of the grammar. This method can not be used in TRANSFORMERS. Currently, the C++ AG of TRANSFORMERS contains more than 1000 strategies defined by the user in order to disambiguate the C++. In addition some production names are generated at the generation of the parse table used by SGLR (Visser, 1997). So the approach used by UU-AG can not be used by TRANSFORMERS following the quantity of symbols.

For example you can see, in Figure 3.1, a small tree which represents the C++ Abstract Syntax Tree (AST) of the code "int main(){return 0;}". Each word in the previous quote has the same color as a tree node in Figure 3.1.

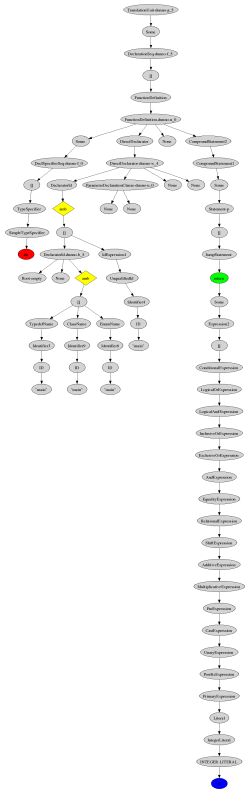

Figure 3.1: “int main(){ return 0; }” C++ AST.

To do this we have to review the problem in detail. To facilitate the process we have to split it in two main parts: the need of propagation, propagation rules. The need of propagation is a simple part which has to be common to all propagations. Propagation rules are used to define how a specific attribute have to be propagated. Propagation rules could be Top-Down (TD), Bottom-Up (BU) or Left-to-Right (LR).

This section deals with the common part the propagation problem.

A way to solve this problem is to find all attributes that are used and that have no definition. If this way still exist after the attribute propagation then the evaluation or compilation process should failed because no value have been associated to this attribute. The goal of the attribute propagator is to prevent that case either by propagating attributes correctly or by throwing an error.

Table 3.2 is used to check all possible cases that can occur in the AG system of TRANSFORMERS. Those cases are combined with a small conclusion which describes the result and what we expect to do with the attribute propagator.

The symbol ✓ specifies that the expression is verified. The symbol ✗ specifies that the expression is not verified. If there is no symbol, the expression result is ignored.

The expressions used in Table 3.2 are the following:

- The ATTRIBUTE "a" is *used* by a STRATEGY which is *on* a PRODUCTION; the PRODUCTION *uses* the SYMBOL *on* which the ATTRIBUTE "a" is.

$$\left\langle U^{\dagger}_{syn} \right\rangle \tag{3.1}$$

- The ATTRIBUTE "a" is *produced* by a STRATEGY which is *on* a PRODUCTION; the PRODUCTION *produces* the SYMBOL *on* which the ATTRIBUTE "a" is.

$$\left\langle P^{\dagger}_{syn} \right\rangle \tag{3.2}$$

- The ATTRIBUTE "a" is *produced* by one STRATEGY *on* each PRODUCTION; each PRODUCTION *produces* the SYMBOL *on* which the ATTRIBUTE "a" is.

$$[O]\left[P^{\dagger}\right]\left\langle O^{\dagger}\right\rangle[P_{syn}] \tag{3.3}$$

- The ATTRIBUTE "a" is *used* by a STRATEGY which is *on* a PRODUCTION; the PRODUCTION *produces* the SYMBOL *on* which the ATTRIBUTE "a" is.

$$\left\langle U^{\dagger}_{inh} \right\rangle \tag{3.4}$$

- The ATTRIBUTE "a" is *produced* by a STRATEGY which is *on* a PRODUCTION; the PRODUCTION *uses* the SYMBOL *on* which the ATTRIBUTE "a" is.

$$\left\langle P^{\dagger}_{inh} \right\rangle \tag{3.5}$$

- The ATTRIBUTE "a" is *produced* by one STRATEGY *on* each PRODUCTION; each PRODUCTION *uses* the SYMBOL *on* which the ATTRIBUTE "a" is.

$$[O]\left[U^{\dagger}\right]\left\langle O^{\dagger}\right\rangle[P_{inh}] \tag{3.6}$$

(3.1)	(3.2)	(3.4)	(3.5)	Conclusion
✗	✓			The attribute is considered as a local attribute.
✓				The Equation 3.2 has to be satisfied.
✓	✓			The Equation 3.3 has to be satisfied.
		✓		The Equation 3.5 has to be satisfied.
			✓	The Equation 3.6 has to be satisfied.
	✓		✓	The attribute cannot be produced as synthesized and inherited.

Table 3.2: Conditions of the propagation system.

This table shows us that we have to compute it each time on each attribute due to the local attribute detection. An ATTRIBUTE can be considered as local while no *use* exists, but the attribute propagator system can add a *use* of this ATTRIBUTE and the ATTRIBUTE won't be local any more.

The implementation of the common part of the propagation uses the previous expressions. There is a small algorithm that compute one step of the propagation. Before going forward we have to remark some important points:

- Each grammar is a finite structure.
- The algorithm never remove information from the grammar.
- The algorithm never add information if it already exists.
- The modification of some parts of the grammar can induce a modification of other parts.

Those remarks enable us to use the algorithm in a fix-point algorithm. So, we are sure that the attribute propagation do the job well and terminate.

3.2 Propagation Rules

A propagation rule is a list of rules that define how the propagator should act when it is trying to propagate an attribute. The goal of this section is to define rules used in BU, TD, LR propagations.

A rule defines a way to add a strategy or information to compute the attribute propagation in the context of a production.

To simplify the problem of specific propagation, we have added the idea of flags. A flag is a mark that is set on a symbol. It specifies that an attribute with a certain name and a certain kind can be added on the symbol. Flags are used to define a cover set for attribute propagation rules. They are mainly used to reduce the propagation problem to a problem that is local to a unique production. Only the BU and LR propagations required the use of these flags.

Paradoxically we do not want to add everything. This restriction comes after an observation of the previous system. The previous system automatically added a BU attribute where it was required but if it can not find any definition it will define its value on the empty list. The problem arise when a propagation problem is not seen by the user and the system cannot warn him. In the new system this problem has been solved by merely activating that kind of propagation on empty productions.

3.2.1 Top-Down

TD attributes don't need flags and they need only one rule to define the TD propagation.

TD attributes are coming from the top to go to the lower part of a parse tree. So any requirement of this kind of attribute will ask the parent symbol in a production to define it.

3.2.2 Bottom-Up

BU flags need two rules.

The first rule is used to define if a symbol possesses a BU attribute. If so, then the symbol should also possesses a BU flag.

The second rule is used to define if a symbol possesses a BU flag. If so, then all its parents in all existing productions should have the same BU flag.

BU attributes need only one rule.

This rule defines that on a production the value is a concatenation of all values coming from all symbols used by the production that has the corresponding BU flag.

3.2.3 Left-to-Right

LR flags need two rules.

LR flags are annotated with a value. This value could be inherited, synthesized or both. The inherited value means that the represented attribute expect to be define. The synthesized value means that the represented attribute is already define.

The first rule is used to define if a symbol possesses a LR attribute. If so, then it should also possesses a LR flag. If the attribute is used then the corresponding flag have the inherited value. If the attribute is produced then the corresponding flag have the synthesized value.

The second rule is a bottom-up propagation of LR flags. That means if in a production some symbols can use or produce a LR attribute then the produced symbol can possibly use or produce the same LR attribute.

LR attributes need only one rule.

This rule requires to have a list of symbols which are in the same order as the production from where they are coming from. This list has to have the produced symbol at the first and at the last position in the list. This list is used to fetch the nearest symbol on the left that satisfies a property. The property checks if it exists an inherited flag on a used symbol or a synthesized flag on the produced symbol. The property is used to fetch a symbol that can produce a value to define the requested attribute.

Chapter 4

Disambiguation

After the generation by SGLR (van den Brand et al., 2002) of an ambiguous parse-tree, we have to disambiguate with our AG system.

In this chapter we are looking for proving that our system is correct. To accomplish this hard task we have to: define what is required to disambiguate; define an ambiguity; define the formal process of our evaluation; and verify the validity of the disambiguation.

4.1 Ambiguities definition

An ambiguities is the possibility to find different interpretations for a unique text at a specific abstraction level. The disambiguation process is the fact to disambiguate something by the use of an abstraction that is over the abstraction level of the ambiguities.

In our case we are trying to disambiguate syntactical ambiguities by the use of semantics' evaluation. We have to understand that we are not able to disambiguate semantics' ambiguities by the use of semantics' evaluation. That is the reason why some ambiguities can remain after our disambiguation process.

4.2 Attribute Evaluation

To prove if we can disambiguate with our current system, we have to check how it proceeds and what its limitations are.

The first step of the evaluation is to add attributes on the parse-tree. The evaluator read the input tree and identifies each node with the production used by the parser. Then it adds attributes on the tree and defines a unique identifier. The identifier represents the future attribute value and refers to a list of functions that are used to compute the value of the attribute.

There are two different attribute values: valid or invalid. In TRANSFORMERS the invalid value is represented as a specific term that should not be used any where else. Otherwise a normal term, as defined in the STRATEGO language, is accepted as a valid value.

We need a list of functions because an attribute can be computed by different ways if its value is synthesized through an ambiguity. The problem with this list of functions is what happens when two or more functions return valid values. This problem is currently solved by comparing all returned values. If all values are identical then the value is synthesized. Otherwise the synthesized value is declared as invalid. This is a dangerous feature of our system.

After the evaluation process, each node of the parse-tree has annotations. Each annotation contains all attributes of the corresponding symbol with their values. At this moments all ambiguities remains in the tree. The last step of this process is to remove all invalid branch from the tree and return the disambiguated tree. This process use a top-down traversal to find any invalid trees.

- If in the current tree root an attribute is invalid then the tree is invalid.
- If in the current tree one of its branches is invalid then the current tree is invalid.
- If the current tree root is an ambiguity node then the process keep all valid trees.
- If there is no more valid trees under an ambiguity node then the current tree is invalid.

4.3 Disambiguation Validity

Now, we will see if our system is able to disambiguate any parse-tree. To achieve this task we have to define what is our problem about the disambiguation process. After we will try to prove that our system is able to disambiguate some minimal cases that are representative of all cases found in some ugly grammars. (**?**)

We have to keep in mind that disambiguate does not mean removing all ambiguities from a tree, but keeping only all ambiguities that could not be solved by the semantics of the language.

All disambiguation problems are caused by declarations and the use of the declared elements. This problem can be seen as a source of information and as an element that required this information. To describe this problem we have four trees where each represents one case of our problem. (see Figure 4.1 to 4.4)

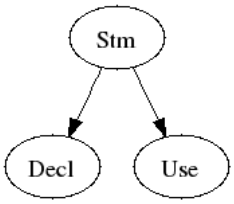

Figure 4.1: No ambiguities case.

Figure 4.1 has no ambiguities[1] and so can be disambiguated. It represents the case where the declaration and the use are not ambiguous.

[1] This is not a joke, we just want to observe all cases

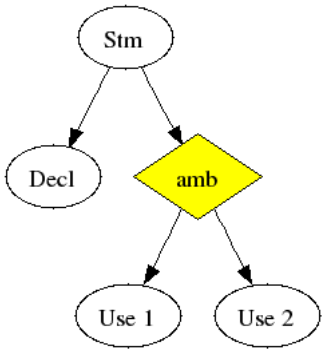

Figure 4.2: Ambiguous Use.

Figure 4.2 has an ambiguity on the use part of the tree. To disambiguate that kind of tree we have to fetch information from the declaration part of the tree. On each branch of the ambiguous node, we check if the use is coherent with the information.

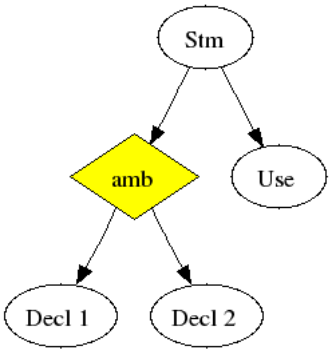

Figure 4.3: Ambiguous Declaration.

Figure 4.3 has an ambiguity on the declaration part of the tree. This tree is more complex to solve than the previous case because you have to remember that the system use the declaration to verify the usage. As describe in our attribute evaluation our system cannot fetch information from an ambiguity. To solve this problem, a special attribute has been introduced. It is named "replace". It allows the user to delay the evaluation by copying the tree and replacing it after. To disambiguate the previous tree we first have to add the tree with the ambiguous node in the environment and check on the usage if one of the declarations can match the usage, in order to be coherent.

This problem can also be solved by using the same method used to disambiguate Figure 4.2. Be careful, you cannot use this method to handle the two cases at the same time without having a cycle in our AG.

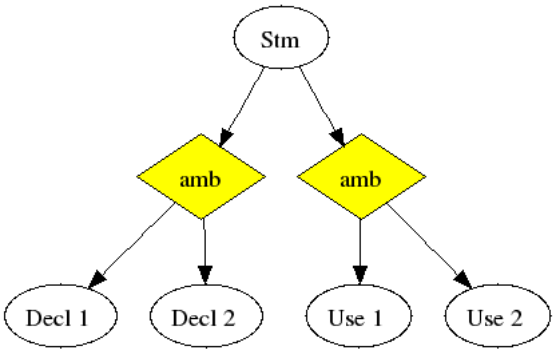

Figure 4.4: Ambiguous Declaration & Use.

Figure 4.4 has ambiguities on the declaration part and on the use part of the tree. The method uses to solved this problem is quite similar to the disambiguation of the declaration. First, we have to add the ambiguous trees corresponding respectively to the declaration part and to the use part. Second, we have to check for each couple of trees if some of them are coherent.

Unfortunately our system is not able to raise an ambiguity over the *Root* node. So, if we have only some couple that are valid like "Decl 1" with "Use 2" and "Decl 2" with "Use 1" then our system will generate a tree that is identical to the input tree (Figure 4.4) but in this case we expect a better disambiguation like Figure 4.5.

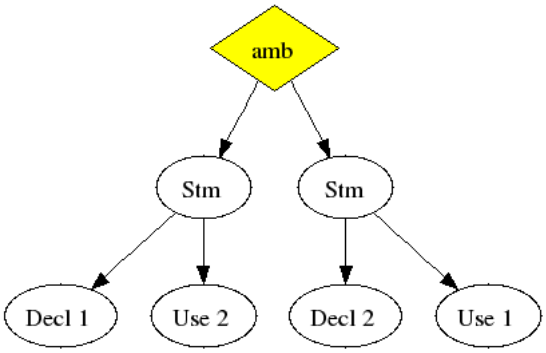

Figure 4.5: Ambiguous Declaration & Use.

This solution is more explicit because it has raised all ambiguous nodes over the *Root* and filtered all valid couples.

So our system is not able to generate a clear output to specify which couples are valid and so it is not able to disambiguate this case correctly except if there is only one solution.

We hope that case does not happen in the C++. This case has already happened when the *typename* keyword was not in C++. The problem with the *typename* keyword was that all template definitions are totally ambiguous because they could be use with different parameters. This keyword removes an ambiguity of the kind seen in Figure 4.4.

Chapter 5

Conclusion

TRANSFORMERS is a project that has been created to transform some C++ code. To accomplish this task we have to disambiguate the C++.

With this, AGs can be manipulated with a new notation. This notation uses the hybrid logic language. The hybrid logic language can be used to express with ease conditions on a graph. As presented, an AG can be represented as a graph. With this new abstraction we are now able to implement correctly conditions. So there are no more differences between a source code and an expression expresses in hybrid logic.

This work has offered the possibility to build another attribute propagator system. It gives us a good background to continue to work with AGs. It has been created to be modular. A first part defines propagation conditions. Propagation conditions define all parts common for all attribute propagations. Others parts define attribute propagations that are very dependent on the attribute kind.

The conclusion of this report is that we are able to disambiguate the C++ except if we found a new ambiguity similar to the problem of the **typename** keyword that implies that we have an ambiguous declaration and an ambiguous use. This work concludes the problem of the disambiguation with our AG system.

Expected future works on the attribute propagator would be to define a language to describe any propagations rules. This system would offer us the possibility to extend our attribute propagation system dynamically. In addition, we can expect to be able to define our own attribute kinds.

Appendix A

Bibliography

Blackburn, P. (2000). Representation, reasoning, and relational structures: A hybrid logic manifesto. *Logic Journal of the IGPL*, 8(3):339–365.

van den Brand, M. G. J., Scheerder, J., Vinju, J., and Visser, E. (2002). Disambiguation filters for scannerless generalized LR parsers. In Horspool, N., editor, *Compiler Construction (CC'02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, Grenoble, France. Springer-Verlag.

Bravenboer, M., Kalleberg, K. T., Vermaas, R., and Visser, E. (2006). Stratego/xt 0.16. components for transformation systems. In *ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM '06)*, Charleston, South Carolina. ACM SIGPLAN. (To appear).

ISO/IEC (1999). ISO/IEC 9899:1999 (e). programming languages — C.

ISO/IEC (2003). ISO/IEC 14882:2003 (e). programming languages — C++.

Kalleberg, K. T. and Visser, E. (2006). Strategic graph rewriting: Transforming and traversing terms with references. In *Proceedings of the 6th International Workshop on Reduction Strategies in Rewriting and Programming*, Seattle, Washington. Accepted for publication.

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

Swierstra, S. D., Baars, A., and Löh, A. (2003). The UU-AG attribute grammar system. `http://www.cs.uu.nl/groups/ST`.

Visser, E. (1997). Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam.

List of Figure

List of Table