Attribute grammars for C++ disambiguation

Valentin David **<valentin@lrde.epita.fr>**

Technical Report n^o0405, December 2004
revision 667

The development of the Transformers project has raised some design isues. Static code for disambiguation is quite hard to write and maintain. Disambiguation and type checking are interdependent. This last problem is a showstopper. The Transformers project needs a new architecture for disambiguation. This technical report is about a possible solution: attribute grammars.

Le développement de Transformers a soulevé certains problèmes de conception. Les programmes statiques pour la désambiguisation sont trop durs à développer et à maintenir. La disambiguisation et l'analyse sémantique sont deux étapes qui sont mutuellement dépendantes. Ce dernier problème bloque le développement du projet. Transformers a besoin d'une nouvelle architecture pour la désambiguisation. Ce rapport technique explique une solution possible: les grammaires attribuées.

Keywords
C++, attribute, grammar, ambiguities, disambiguation

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
lrde@lrde.epita.fr – http://www.lrde.epita.fr/

Copying this document

Copyright © 2004 LRDE.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being just "Copying this document", no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is provided in the file COPYING.DOC.

Contents

1 Introduction **8**
1.1 Disambiguation 8
1.2 Problems 8
1.3 Attribute grammars 8
1.4 C++ disambiguation 9
1.4.1 Local disambiguations 9
1.4.2 Contextual disambiguations 9

2 Attribute Grammars **11**
2.1 Attributes 11
2.2 Implementation concepts 11
2.2.1 Evaluator 11
2.2.2 Checkers 12

3 `sdf-attribute` **13**
3.1 Principles 13
3.2 Syntax 13
3.3 Attributing rules in the parse table 14
3.4 Definition check 16
3.5 Evaluation 16

4 Disambiguation **18**
4.1 Principles 18
4.2 Local disambiguation 19
4.3 Contextual disambiguation 21
4.4 Ambiguous synthesizing 22
4.5 Results 22

5 Templates **24**
5.1 Ambiguities in template definitions 24
5.1.1 Type names 24
5.1.2 Nested names 24
5.1.3 No ambiguity 25
5.2 Ambiguities outside template definitions 26
5.3 Attribute grammars and templates 27
5.3.1 Function evaluation and attribute grammars 28
5.3.2 Template definitions and attribute grammars 29
5.4 Fit with the standard 32

5.4.1 Multiple delay layers in instantiation 34
5.4.2 Disambiguation inside of template definition 34

6 Conclusion 36

A Mini C++ 37

B Bibliography 49

List of Figures

3.2 AsFix tree node . . . 14
3.4 Parse table generation . . . 15
3.5 More accessible attributes . . . 16
3.6 To AsFix . . . 17
3.7 Evaluation process . . . 17

4.1 Evaluation process . . . 19
4.3 Template parameter ambiguity . . . 20
4.6 Namespace definition ambiguity . . . 22

5.12 Attribute dependencies for template definition . . . 32

List of Syntax Definitions

3.1 SDF production rule 13
3.3 Example of attributes 14
4.2 Template parameter ambiguous production rules 19
4.4 Template parameter disambiguation 21
4.5 Namespace ambiguous production rules 21
4.7 Namespace disambiguation 23
5.9 Example of attribute rules for function definition for evaluation 30
5.10 Example of attribute rules for function evaluation 30
5.11 Example of template definition disambiguation 31
5.13 Example of template specialization disambiguation 32
5.14 Example of template specialization disambiguation 33

List of Programs

1.1 Currently not resolvable program 10
5.1 Syntactic ambiguity in a template, but `t` must be a value. 25
5.2 Keyword for explicit disambiguation 25
5.3 Ambiguity on `nested-name-specifiers` 25
5.4 `typedef-name` and `class-name` 26
5.5 Ambiguities on template instantiation. 27
5.6 Recursive template definition 27
5.7 Obfuscated recursive template definition 28
5.8 Instantiation of members 29
5.15 Member instantiation 34

Chapter 1

Introduction

Transformers is a framework for C++ program transformation. Most of the current work has been done on parsing (and pretty-printing, that is the inverse operation), not on the transformation themselves (Anisko et al., 2003).

Transformers uses a generic rewriting framework called Stratego/XT, initiated by Visser (2001). It uses SGLR[1] parser, which supports all context-free syntaxes. However, C++ is not context free.

The approach used in Transformers for parsing is to use the ambiguous context-free syntax provided by the C++ standard (C++ standard, 2003), along with desambiguation filters.

1.1 Disambiguation

SGLR parsing handles ambiguous grammar: the output tree contains some ambiguity nodes that propose several subtrees to interpreting a subpart of the source code differently. The disambiguation consists in pruning bad interpretation branches.

1.2 Problems

Currently, disambiguation filters are static programs written in Stratego. There is a conceptual problem. One desired capability with Transformers is to extend the syntax of C++. The problem is that some disambiguation filters need knowledge to resolve contextual ambiguities. Modifying the disambiguation filters to support extensions is not an easy task.

1.3 Attribute grammars

Attribute grammars consist in adding code for parse tree annotating in the grammar definition. This technical report shows how this technique can resolve our ambiguity problems.

[1] Scanner-less Generalized LR.

1.4 C++ disambiguation

Currently, disambiguation is made by some static filters written in Stratego. There are many kinds of ambiguities in C++, separated into two categories: local or pseudo-local, and contextual. The local disambiguation filters for local ambiguities do not need the whole context, only a localized one, whereas contextual disambiguation need tables to register the quality of each object found during the tree traversal.

1.4.1 Local disambiguations

An example of local disambiguation is the template parameters one. Currently, if you try to parse `class I` as a template parameter, the syntax can interpret it into two different ways. The first one is the human way to read it. It is interpreted like `typename I`. But we can also read it, according to the syntax definition, like `class I i`, where `i` was forgotten because unused.

The C++ standard, in the text body, specifies that a non-type template parameter cannot be a class instance. So we can delete all branches that do not obey to this rule and then disambiguate this way.

1.4.2 Contextual disambiguations

Some other disambiguation filters need more context to work. Disambiguations need name resolution.

Namespaces

When declaring a non-anonymous namespace, there are two ways to read it. Either it was not already defined, and it is an *original namespace definition*, or it was already defined, and it is an *extension namespace definition*. This disambiguation remains simple since it needs only context carrying on namespace definitions.

Type categories

A major part of disambiguation is brought with type kinds. We have several categories of types, mainly: typedef name, class name and enum name. In most of type uses, there is an ambiguity with these three categories. It has to be resolved using the context. This disambiguation is further complicated because of namespaces and scopes.

Problem with templates

Because of explicit template instantiations, we need more semantic analysis than a simple kind checker. We also need a part of the type checker. The program 1.1 is not resolvable until we have type checking and we do not know which specialization of the class template we use. The problem is that the type checking must be done after the disambiguation. In fact, the two passes are interdependent.

```
template <typename I>
class A {
  typedef int t;
};
template <>
class A<int> {
  static int t;
};
void f() {
  int f(A<int>::t); // is t a type or a variable?
}
```

Program 1.1: Currently not resolvable program

Chapter 2

Attribute Grammars

Attribute grammars, introduced by Knuth (1968), is a way for parse trees annotating. It permits to express context handling into the grammar.

Each node of the tree has attributes that are computed from the father node or child nodes. The way to compute these attributes is described with rules. So these rules are attached to the syntax rules. With these rules, we know the dependencies between attributes. We can deduce a way to evaluate the value of each attribute. Commonly, it consists of a topological sort of the dependency graph.

After building the parse tree, an attribute evaluator traverses it so as to calculate all the attributes. The evaluator can be either dynamic, or generated by an evaluator generator.

2.1 Attributes

There are two kinds of attributes: inherited and synthesized. It is a convention. Inherited attributes are calculated from the upper node and synthesized ones are calculated from child nodes.

2.2 Implementation concepts

There are two kinds of tools to develop for attribute grammars handling. First, an evaluator is needed. It is the main part. But it can fail if the grammar is invalid. We need to detect errors in the grammar. It is the job of the checker.

2.2.1 Evaluator

The attribute evaluation depends on each other. The evaluation needs to sort the attribute evaluations. There are several techniques for evaluation:

- Build a dependency graph and then to make a topological sort. Then, the evaluation only needs to follow this way to evaluate each attribute in the tree.
- Traverse the tree with conditional directions. If a new attribute is computed, children must be traversed again.

- Computing attributes partitions statically is a better solution. Each partition corresponds to an evaluation pass. Then, the evaluation consists of a statically known number of pass.

2.2.2 Checkers

Two kinds of errors can be made with attribute grammars. Either some attributes are not well defined, or there are some cycles in the dependency graph.

Definition

Each attribute depends on other ones. On a context-free grammar, rules can produce a non terminal symbol. It is easy to forget writing a rule that produce a needed attribute. One of the checks is to verify that all needed attributes will be necessarily calculated by any produced tree.

Cycle detection

Since we need a topological sort of our dependency graph, it is implied that there must be no cycle in it. In fact, there is no way to evaluate attributes in a cycle.

The cycle detection consists in a transitive closure. When a cycle exists, there are local cycles on this closure.

Chapter 3

sdf-attribute

`sdf-attribute` is a tool bundle to use attribute grammars in the Stratego/XT environment. It is divided in two parts: the table builder and the tree evaluator.

3.1 Principles

In the Stratego/XT environment, the parser is SGLR. It needs a parse table. This parse table is computed from a syntax definition written in SDF. An SDF definition file is composed by several production rules. Each production rule has a list of children symbols, the produced symbols, and a list of annotations as described in figure 3.1.

```
Some Symbols → Produced {cons("Constructor")}
```

Syntax definition 3.1: SDF production rule

These rules and theur annotations are kept in the parse table when building it. They are reused when building a parse tree. Production rules are attached on it. Parse trees are represented in AsFix format. This format consists of `appl` nodes that have two children. The first one is the production rule used and the second one is the list of the `appl` child nodes. This is described in figure 3.2.

So, the annotations are attached to the production rule descriptions. Annotations can be kept up to the parse tree production. `sdf-attribute` uses this feature: attribute rules are kept into the annotation lists. Then, the evaluation only has to traverse the tree and evaluate attributes.

3.2 Syntax

Each attribute has a name and a name space name. It is written like `NodeName.namespace:name`. The name space name is optional, defaulting to the one specified for the rule list.

For each syntax production rule, lists of attribute rules can be specified in the annotations. The first one specifies the default name space, then the list of rules.

A rule defines the way to evaluate an attribute with a syntax similar to `Node.attr := strat`. A strategy in Stratego computes the attribute value. Attributes are used into strategies like Stratego variables.

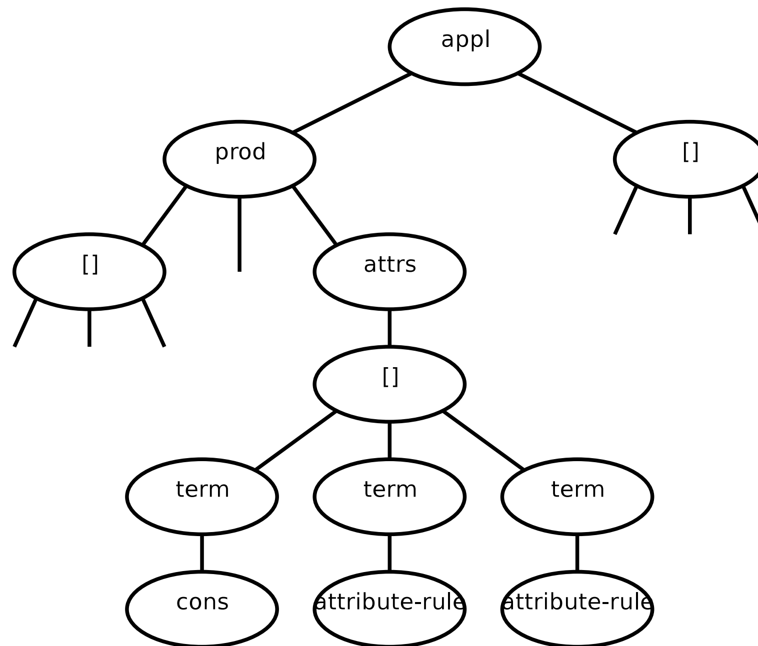

Figure 3.2: AsFix tree node

The node name is either `root`, when we talk about 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 a same symbol name is used twice or more in the same rule. An example is given in syntax definition 3.3.

```
"namespace" Identifier "{" NamespaceBody "}"
                            → OriginalNamespaceDefinition
  {attributes(namespace:
   root.name                := <concat-strings> [root.namespace,
                                                 Identifier.string, "::"]
   NamespaceBody.namespace  := !root.name
  )}
```

Syntax definition 3.3: Example of attributes

3.3 Attributing rules in the parse table

First, a desugaring pass is performed on the syntax definition. This pass consists in adding implicit name space names. When desugaring is done, checks are performed as described in the section 3.4. Original SDF syntax accepts ATerms into annotation lists. In order to keep it parsable to the parse table generator, the attributes are kept in ATerm format. This means that the stratego AST is unchanged when pretty printed. The problem is that the ATerm syntax

described into the SDF syntax forbids to begin an ATerm constructor with an upper case letter. This problem is due to the implicit symbols when using upper case letters. To avoid this problem, a filter adds a lower case letter to all constructors for the parse table generation. Label names are not kept in the parse table. Since they are used during the evaluation, we need to keep it. An alias table is hence added to the annotations. Then, the normal parse table generator is used. The generated parse table is a binary ATerm file. A final pass deletes all beginning lower case letters from constructors to have a cleaned up table. All these stages are shown in figure 3.4.

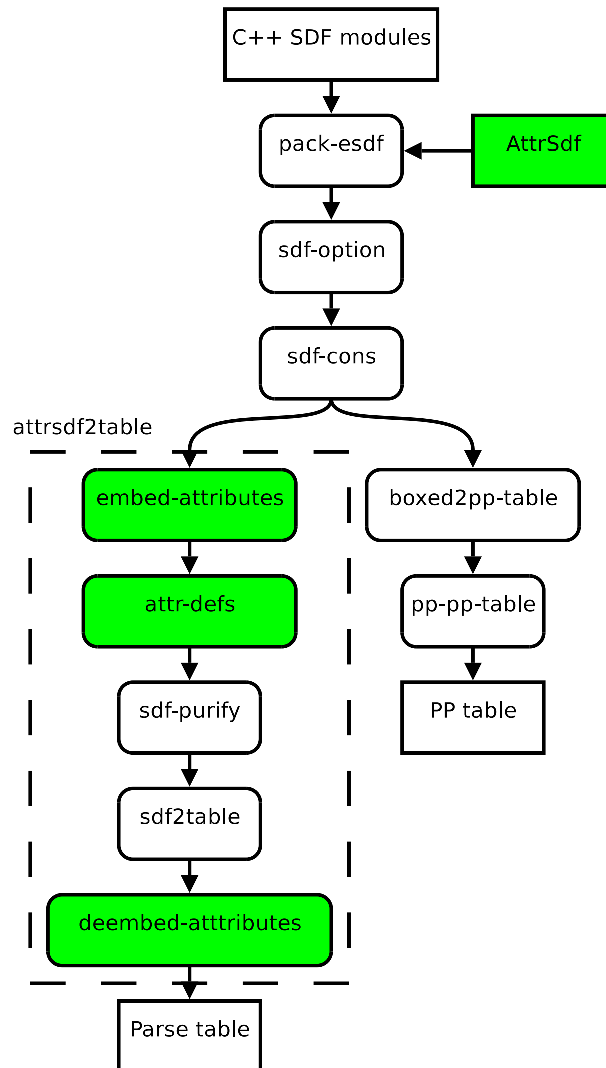

Figure 3.4: Parse table generation

3.4 Definition check

Before generating the table, it is useful to check if the attribute rules are well written. Since there is no debugging tool it is better to find errors instead of creating an invalid table. It is not so hard to find why an attribute value is not good, but it is hard to find why an attribute cannot be evaluated.

The definition checker traverse the graph of all possible trees carrying the knowledge of what was calculated or not beginning with the start symbols.

3.5 Evaluation

After using SGLR with this new parse table to parse a source file, the AsFix tree has attribute rules and label tables as production annotations. So as to make evaluation easier, these informations are moved to a more accessible place as shown on figure 3.5. A empty list is added to accept future attribute values.

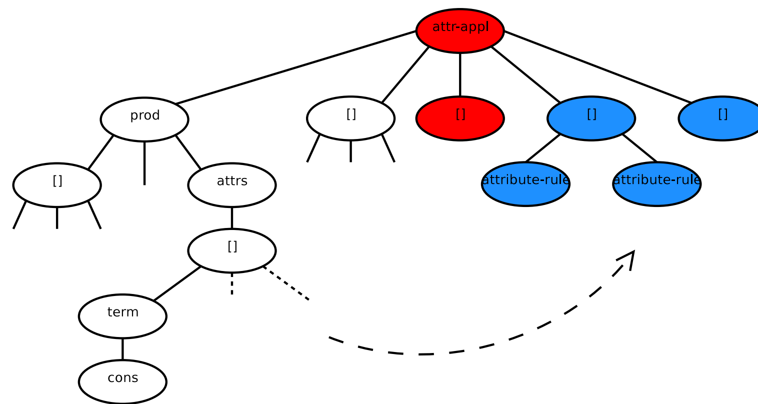

Figure 3.5: More accessible attributes

This tree is then evaluated with the evaluator. The Stratego interpreter library from Stratego-Shell (Bravenboer, 2003) is used so as to evaluate the attributes values. The input term of the strategy is the local node. It permits to extract what the user wants like implode it into a string.

The tree traversal is conditioned with what is computed. When new attributes are evaluated, we can follow again children.

When the tree is evaluated, it can be transformed again into a AsFix format tree as shown on the figure 3.6. The attribute values are put into production rule annotations.

ATerm provides annotation capabilities. This tree can be imploded into an AST with attribute values as annotations. Of course, these values can be deleted at any time with filters. The evaluation process is described in the figure 3.7.

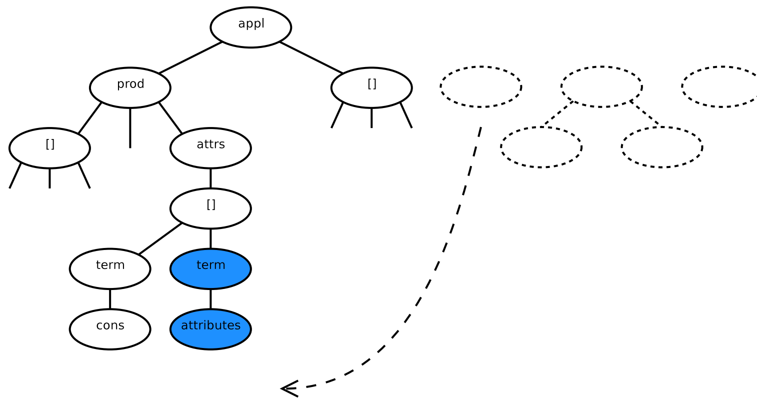

Figure 3.6: To AsFix

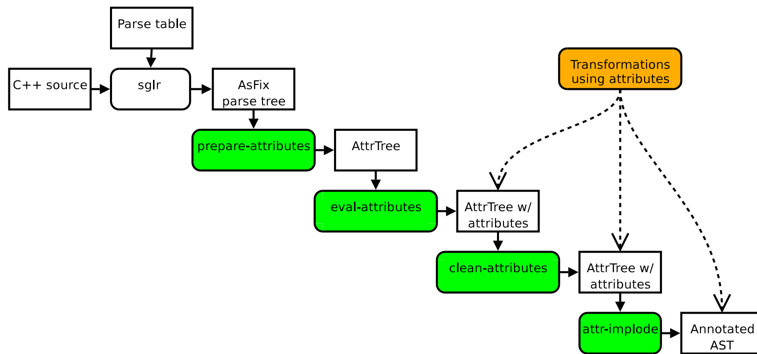

Figure 3.7: Evaluation process

Chapter 4

Disambiguation

The C++ syntax used in Transformers is the standard grammar. This grammar is just a support of the real C++ grammar. In fact, C++ is a context-dependant language. But it is easier to manipulate context free syntaxes. The syntax used is context free but ambiguous. There are some ways to interpret the code with the given syntax. We have to use the context to select the right interpretation.

For a while, Transformers used static programs to disambiguate programs (this is still the case today and this is problematic). Many conception problems were raised up. It is not so natural to write contextual static filters.

A good way to express context into context free grammars is to use attribute grammars. Attribute grammars are useful for type checking or other annotating processes that need context.

This chapter shows how attribute grammars can be used for disambiguation.

4.1 Principles

The global process of disambiguation in Transformers is kept as is. First, we use SGLR to parse and obtain an ambiguous parse forest. Then we try to prune it to get a single unambiguous tree.

The pruning pass is separated in two different treatments. First we evaluate the attributes, and then we find with these attributes how to cut bad branches. In fact bad branches are marked with attributes.

On ambiguities, there is at most one valid branch. There are programs correct syntactically but not semantically. But we can mark enough branches as bad to get a single tree. Using rules from the the standard, we can figure out where the program is invalid. Disambiguation consists in marking the proper place as bad. Good place means that it can reduce the ambiguous tree.

So, the two processes are performed sequentially as shown in figure 4.1. Then the attributes can be removed and we get a disambiguated parse tree.

In Transformers, there are several filters to make this. Two of them will be shown as examples. These two examples figure out all the needs for the C++ disambiguation. Some disambiguations do not need all the context but only a local one. Some others need all the context with symbol tables.

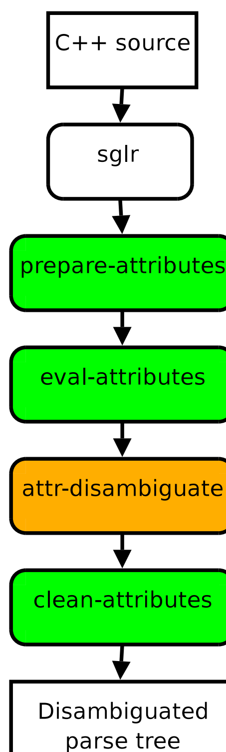

Figure 4.1: Evaluation process

```
TypeParameter                          → TemplateParameter
ParameterDeclaration                   → TemplateParameter

"class" Identifier?                    → TypeParameter

DeclSpecifierSeq AbstractDeclarator?   → ParameterDeclaration
TypeSpecifier                          → DeclSpecifier
ElaboratedTypeSpecifier                → TypeSpecifier
ClassKey "::"? NestedNameSpecifier? ClassName
                                       → ElaboratedTypeSpecifier
"class"                                → ClassKey
Identifier                             → ClassName
```

Syntax definition 4.2: Template parameter ambiguous production rules

4.2 Local disambiguation

The first disambiguation shows filter ambiguities on template parameters. In the C++ syntax, a template parameter can be either a type parameter or a parameter declaration (see syntax definition 4.2). But when the parameter to parse is something like `class I`, then it can be interpreted as two different trees shown in the figure 4.3. According to the C++ standard paragraph 14.1.6,

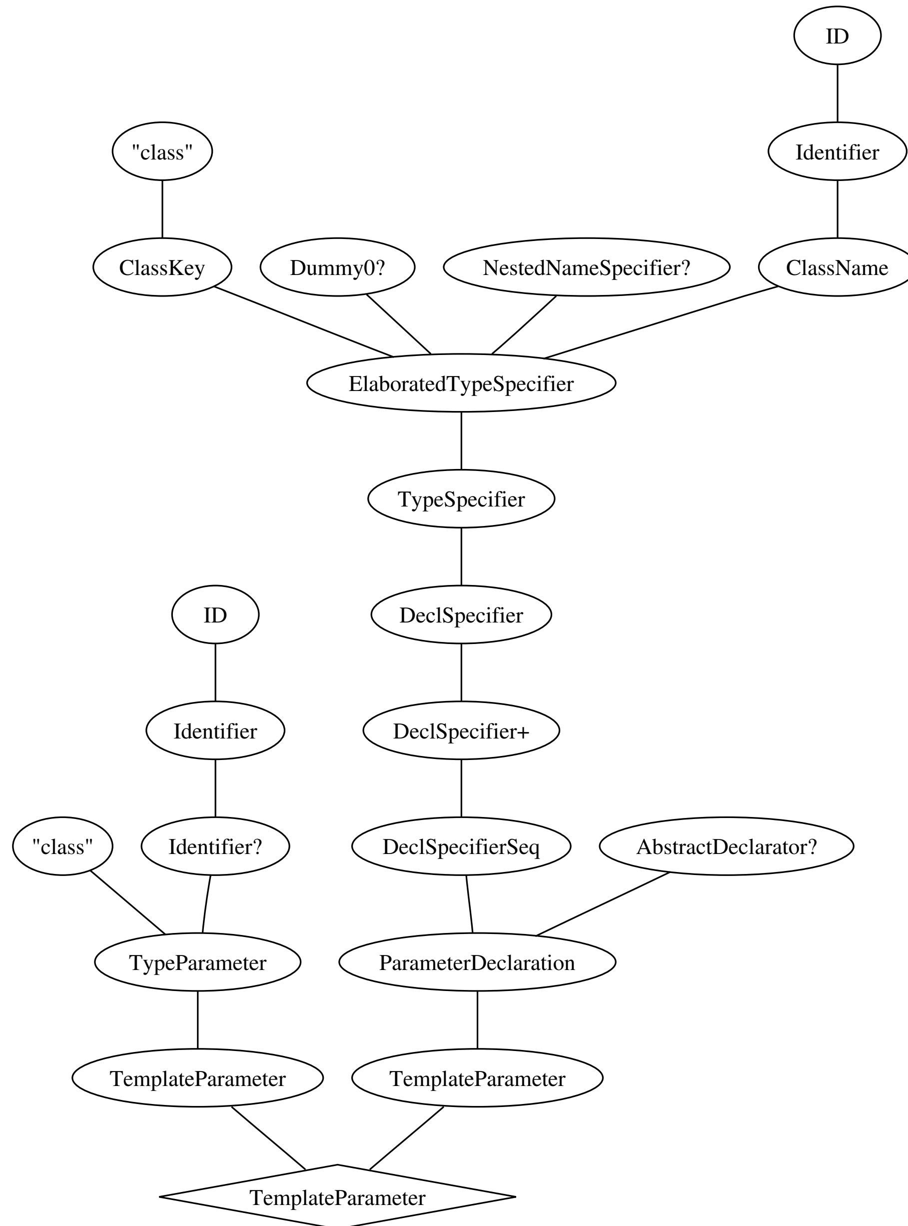

Figure 4.3: Template parameter ambiguity

a class instance cannot be passed as a template parameter. So we know that the solution is to see it as a type parameter.

To resolve this ambiguity, we just have to propagate an inherited attribute that authorizes to use a class instance as parameter or not. The syntax definition 4.4 shows some of the used rules. Note that some other rules are needed to carry attributes values to descendant nodes.

The `ok` attribute is the mark for disambiguation. If it contains `0`, it is false so, the correspond-

```
"class" → ClassKey
  {attributes(classparam:
   root.ok := !root.authorize_class
  )}
ClassKey sep:"::"? nns:NestedNameSpecifier? ClassName
                          → ElaboratedTypeSpecifier
  {attributes(classparam:
   ClassKey.authorize_class :=
     !nns.opt; 1; !sep.opt; 1 < !root.authorize_class + !1
  )}
ParameterDeclaration → TemplateParameter
  {attributes(classparam:
   ParameterDeclaration.authorize_class := !0
  )}
ParameterDeclaration → {ParameterDeclaration ","}+
  {attributes(classparam:
   ParameterDeclaration.authorize_class := !1
  )}
DeclSpecifierSeq ad:AbstractDeclarator? → ParameterDeclaration
  {attributes(classparam:
   DeclSpecifierSeq.authorize_class :=
     !ad.opt; 1 < !root.authorize_class + !1
  )}
```

Syntax definition 4.4: Template parameter disambiguation

ing branch will be deleted.

This example shows how a pseudo-local disambiguation is easy to handle with attribute grammars.

4.3 Contextual disambiguation

Another kind of disambiguation is needed for C++. It is a contextual one. An easy fully contextual example of ambiguity is the problem of namespaces. In C++, you can extend an existing namespace definition. In the syntax, the difference has to be expressed as shown in the syntax definition 4.5. The tree ambiguity is shown at the figure 4.6.

```
NamedNamespaceDefinition   → NamespaceDefinition
UnnamedNamespaceDefinition → NamespaceDefinition

"namespace" Identifier "{" NamespaceBody "}"
                           → OriginalNamespaceDefinition
"namespace" OriginalNamespaceName "{" NamespaceBody "}"
                           → ExtensionNamespaceDefinition
```

Syntax definition 4.5: Namespace ambiguous production rules

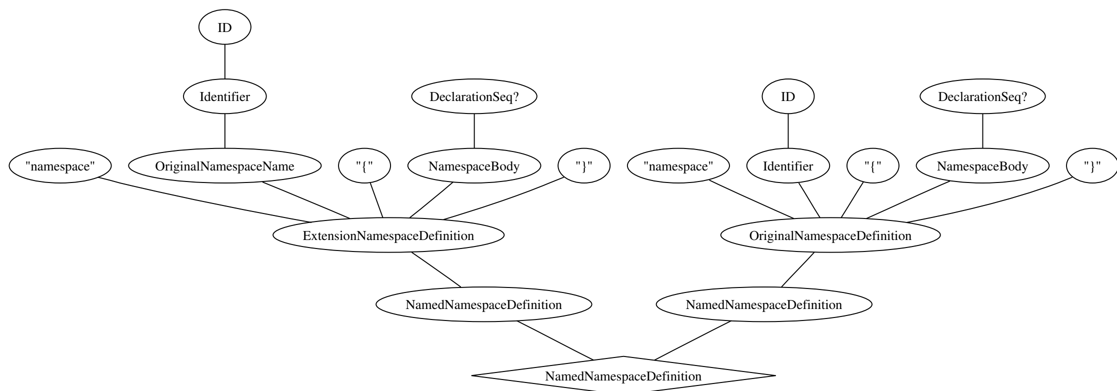

Figure 4.6: Namespace definition ambiguity

Here we have to carry the complete context of namespaces. We use a symbol table that is simply a list of already traversed namespace definitions and the name of the current one. We need the current namespace name too. Since the table has to traverse all the program, we use two attributes: one for inheritance and another of synthesizing.

With this, we can determine if a namespace definition is false like it is made in the syntax definition 4.7.

Since attribute grammars work fine with context (e.g. it is well used for type checking), contextual disambiguation does not bring problems. The resolution remains simple.

4.4 Ambiguous synthesizing

When synthesizing attributes from ambiguities, if the attributes are different, there is no way to determine which one is the right one.

In fact, when we observe what is synthesized on ambiguities (and that can be different), only symbol tables are problematic. However, these tables are carried everywhere, including the node where we can determine if we are in a false branch. Now we know that, we just have to return a special ATerm that the evaluator understands as a bad value. Here we use `invalid()` as described in syntax definition 4.7. This term is understood by the `sdf-attribute` tools as a special one. It means to force to take other values.

4.5 Results

Every C++ ambiguity can be easily handled (compared to static programming) with attribute grammars. It is simpler to write and maintain. With this, syntax extension is easier to write. The filters do not need to be rewritten.

Since type checking can be written with attribute grammars, we can imagine that the disambiguation use it to resolve the problem of template specializations. In fact, attribute grammars help to merge passes. Here, the system of dependency will resolve itself how to traverse the tree and how to combine passes without the user needing to explain it.

The attribute grammars were successfully used for C99 with the presented technique. With C++, the only different point from the C language are the templates which are treated in the next chapter.

```
"namespace" Identifier "{" NamespaceBody "}"
                           → OriginalNamespaceDefinition
  {attributes(namespace:
   root.name               := <concat-strings> [root.namespace,
                                                Identifier.string, "::"]
   NamespaceBody.namespace := !root.name
   NamespaceBody.table     := !root.table
   root.ok                 := <lookup> (root.name, root.table)
                                 => original-namespace-name(); !0 <+ !1
   root.new_table          := !root.ok; 1
                                  < ![(root.name,
                                       original-namespace-name())|
                                         NamespaceBody.new_table]
                                 + !invalid()
  )}

"namespace" OriginalNamespaceName "{" NamespaceBody "}"
                           → ExtensionNamespaceDefinition
  {attributes(namespace:
   root.name               := <concat-strings>
                                    [root.namespace,
                                     OriginalNamespaceName.string,
                                     "::"]
   NamespaceBody.namespace := !root.name
   NamespaceBody.table     := !root.table
   root.ok                 := <lookup> (root.name, root.table)
                                => original-namespace-name()
                             ; !1 <+ !0
   root.new_table          := !root.ok
                             ; 1 < !NamespaceBody.new_table + !invalid()
  )}
```

Syntax definition 4.7: Namespace disambiguation

Chapter 5

Templates

The problem of template disambiguation is big enough to deserve its own chapter. The C++ language, before being normalized, has evolved through the different compiler implementations. It has picked many little things up that imply the way to implement the parser, in fact, the same way that what has been done. But using attribute grammars was not so natural because it is not "industrial" enough. There are many weird behaviors of the language with templates. These behaviors were due to the implementation done. Then, to follow the standard, we must implement these ones which is not very natural for attribute grammars. This chapter presents how the implementation of template disambiguation with attribute grammars can be done.

5.1 Ambiguities in template definitions

5.1.1 Type names

Template code has not all the context provided. This code has to be "intantiated". It means that it is when it will be used that we will know to which kind (e.g. `typedef`, `class`, `namespace`, etc.) all symbols will be mapped. If you see program 5.1, then you will ask yourselves of what the kind the `t` member of `T` is. If the standard did not handle these case, there would be no way to disambiguate until we use it. But it would need to make non linear disambiguation which was impossible with a classical approach.

To make it easier, the standard introduced the keyword `typename` as figured out in program 5.2. It is used to manually disambiguate. Note that `class`, `struct`, `union` or `enum` can be used. When the symbol depends on an argument, then it is a variable if keyword `typename` is not specified like in program 5.1. If the `typename` is used it is then a `elaborated-type-specifier` which is itself a `type-specifier`.

5.1.2 Nested names

Symbols which depends on arguments are unknown. It is the same with `nested-name-specifiers`. But when reading the standard grammar, we can discover that a `nested-name-specifier` can only be `namespace-name` or a `class-name`. Since namespaces cannot be passed through template parameters and classes cannot contain a namespace, these depending `nested-name-specifiers` can only be `class-names`. In program 5.3, there is no way to interpret `n` as `namespace-name`. It is a `class-name`.

```
template <typename T>
struct A {
  static void g() {
    int f(T::t);
  }
};
```

Program 5.1: Syntactic ambiguity in a template, but `t` must be a value.

```
template <typename T>
struct A {
  static void g() {
    int f(typename T::t);
  }
};
```

Program 5.2: Keyword for explicit disambiguation

```
template <typename T>
struct A {
  static void g() {
    int f(typename T::n::t);
  }
};
```

Program 5.3: Ambiguity on `nested-name-specifiers`

Of course, a symbol defined by a `typedef` declaration can be used but it is a `class-name`. In program 5.4, `alias` is a `class-name`.

The C++ standard says that a `typedef-name` that refers to a `class-name` or an `enum-name`, is not really a `typedef-name` but a `class-name` or an `enum-name`. Others are real `typedef-name`.

5.1.3 No ambiguity

All ambiguities that would need to wait instantiation to be resolved are handled by the standard. There is a way to disambiguate directly with the current context, moreover it requires the user to write some extra informations.

```
template <typename T>
struct A {
  static void g() {
    int f(typename T::alias::t);
  }
};

struct B {
  struct T {
    typedef int t;
  };
  typedef T alias;
};

void h()
{
  A<B>::g();
}
```

Program 5.4: `typedef-name` and `class-name`

5.2 Ambiguities outside template definitions

The disambiguation is made easier but all is not simplified. Instantiations have to be made. In program 5.5, `g` and `h` have different kinds, although their definition are pretty much the same. `A<B>::t` is an expression, so `g` will be a variable, whereas `A<C>::t` is a type and `h` is a function.

Here, the standard does not require the user to put a `typename` keyword to disambiguate. The template must be instantiated. In a classical approach, we would carry a template tree (just like an abstract syntax tree) and replace parameters by local arguments. But we cannot really do this with attribute grammars because we would duplicate write all rules twice, one with attribute rules, and another with tree transformation.

```
template <typename T>
struct A {
  typedef T t;
};

struct B {};

template <>
struct A<B>
{
  static int t;
};

struct C {};

void f()
{
  int g(A<B>::t);
  int h(A<C>::t);
}
```

Program 5.5: Ambiguities on template instantiation.

```
template <int I>
struct A : public A<I-1> {
};

template <>
struct A<0> {
  typedef int t;
};

void f()
{
  int g(A<5>::t);
}
```

Program 5.6: Recursive template definition

5.3 Attribute grammars and templates

The problem of templates is very close to function evaluation. We want to evaluate the tree for different parameter contexts.

```
template <int I>
struct A : public A<I-1> {
};
template <>
struct A<3> {
  static int t;
};
template <>
struct A<0> {
  typedef int t;
};
void f()
{
  int g(A<5>::t);
  int h(A<2>::t);
}
```

Program 5.7: Obfuscated recursive template definition

5.3.1 Function evaluation and attribute grammars

One fundamental rule of attribute grammars is that each attribute will be computed once and only once. We can do many things but not bypass this.

We can imagine to do function evaluation in two passes. First, we collect all uses. Then the second pass computes a set of values according to a set of different paramaters. But this will avoid to use recursion. Because some other context will appear on the second pass. Since the number of passes is fix with attribute grammars, this technique disallows recursion. But templates can use recursion because of explicit specialization (see program 5.6). We cannot use it.

Actually, we want to evaluate some subtrees several times according to different contexts. This is not impossible. Nothing forbids to work on attributed trees into the attribute rules.

Adapting the attribute grammar system

To do this we need some extra features:

- Be able to get a tree.
- Add or get attribute values on it.
- Launch the evaluation.

Of course, the evaluation of the subtree must be delayed. For this, we force some trees to not be set. Then, with the dependency rules, the tree is only evaluated partially.

To implement this, a new primitive strategy will be needed. This primitive asks the evaluator to call itself on a subtree. To help, there are some new strategies to manipulate attributes on the trees.

```
template <typename T>
struct A {
  static void g()
  {
    int f(T::t);
    (void) f;
  }
  static void h()
  {
    int f(typename T::t);
  }
};

struct B {
  typedef int t;
};
struct C {
  static int t;
};
void f()
{
  A<C>::g();
  A<B>::h();
}
```

Program 5.8: Instantiation of members

But the attribute grammar has to be checked. The attributes that will be set manually during the evaluation need to have no rules. For checking purpose we add fake rules that do nothing but are only used in the checkers.

All of these allow us to write function evaluation with attribute grammars easily as shown by the syntax definitions 5.9 and 5.10. On declaration, we just add the subtree in the symbol table. Here the expression will not be evaluated until `root.args` is set because its symbol table is dependant on it. Note there is a fake rule to compute this attribute. It is, as said, for the checker. To evaluate the tree to know the value of the expression we will have to set the attribute before evaluate the subtree.

When a function is used, the subtree has to be checked out from the symbol table. On syntax definition 5.10, `root.function_tree` contains this tree. Then, the missing attribute (i.e. `args`) is set before evaluate the subtree. If the grammar was checked as valid and the missing arguments are all set, the evaluation will be complete. Then the value can be picked up from the `eval_exp` attribute.

5.3.2 Template definitions and attribute grammars

For templates it is quite the same. Syntax definitions 5.11, 5.13 and 5.14 show an example of implementation. Here the syntax is simplified to ease the comprehension. In C++, the context for instantiation is the one used with the context linked to the type argument at the template

```
%% Function declaration
"function" Id "(" fas:{FArg ","}+ ")" = Exp -> FunDec
  {attributes(eval:
    %%Map the identifier to the function tree.
    root.out_table := ![(Id.name, <id>)|root.in_table]

    %% The input table is composed but the local table and
    %% arguments.
    Exp.in_table    := <conc>(<zip>(fas.names, root.args), root.in_table)

    %% Result of the evaluation.
    root.eval_exp   := !Exp.value

    %% Warn the checker that it is normal that we
    %% cannot compute this attribute.
    root.args       := extern
  )}
```

Syntax definition 5.9: Example of attribute rules for function definition for evaluation

```
%% Function call
Id "(" es:{Exp ","}* ")" -> Exp
  {attributes(eval:
    %% Get the function tree from the table.
    root.function_tree := <lookup>(Id.name, root.in_table)

                       %% First, add the missing attribute (args).
    root.value           := <add-attribute>
                              (root.function_tree, "eval", "args", es.exp_list)
                       %% Then, call the evaluator on this tree.
                          ; attr-eval
                       %% And finally get the result in the right attribute.
                          ; <get-attribute>(<id>, "eval", "eval_exp")
  )}
```

Syntax definition 5.10: Example of attribute rules for function evaluation

instantiation. Then the symbol table produced by the instantiation is added to the symbol table from the tree at the place of this instantiation.

Syntax definition 5.11 describes the template definition. There are two output symbol tables: one for the definition itself (`out_table`), the other for the instantiation (`specialized_out_table`). Evaluation of attributes `param_type` and `extra_table` are here delayed until it is instantiated. They are the input values for the instantiation. Since `out_table` does not depend on these attributes, the evaluation will be done. Of course before sending the tree to the `out_table`, we have to check that all the inherited attributes that are not delayed have been

set because we will not be able to set it afterward. Since the computation of `out_table` depends on all these inherited attributes (`in_table` and `ns`), we are sure that their rules will be evaluated before the rule of `out_table`. So here we are sure that the tree will be evaluated. Figure 5.12 shows dependencies between attributes.

```
"template" "<" "typename" tn:Id ">" "struct" sn:Id "{" ds:Decl* "}" ";"
  -> Decl
  {attributes(disamb:
    %%Delayed attributes:
    %%   Arguments
    root.param_type   := extern
    %%   Symbol table containing arguments and their members
    root.extra_table  := extern

    %%Computation of the new namespace
    ds.ns      := ![Specialized(sn.string, root.param_type)|root.ns]

    %%Input table for member declarations
    ds.in_table := <conc> (<map(\ (l, t)
                         -> (<conc> (l, [Type(tn.string)]), t) \)>
                         root.extra_table,
                       root.in_table)

    %%Output symbol table before instantiation
    root.out_table  := ![([NotSpecialized(sn.string)|root.ns]
                      , TClass(<id>))
                      |<not(?invalid())> root.in_table] <+ !invalid()

    %%Output symbol table due to the instantiation
    root.specialized_out_table  := !ds.out_table => invalid()
                                   <+ <diff> (ds.out_table, ds.in_table)
  )}
```

Syntax definition 5.11: Example of template definition disambiguation

On specializations, it is simpler. It is like a simple class definition. Syntax definition 5.13 describes it. We traverse the child declarations with the local context and we just declare the specialization in the table.

When using a template we have first to verify if it was already specialized. If not, we instantiate it. It is shown in syntax definition 5.14. `new_instance` will contain the new instance. To instantiate we have to get the tree of the definition. Then we add the missing attributes `param_type` and `extra_table`. And evaluate it. After this, attribute `specialized_out_table` of this tree will contain what we need to add into the symbol table for the instantiation. We can note that `extra_table` will contain only the part of the table that concerns the arguments and its members.

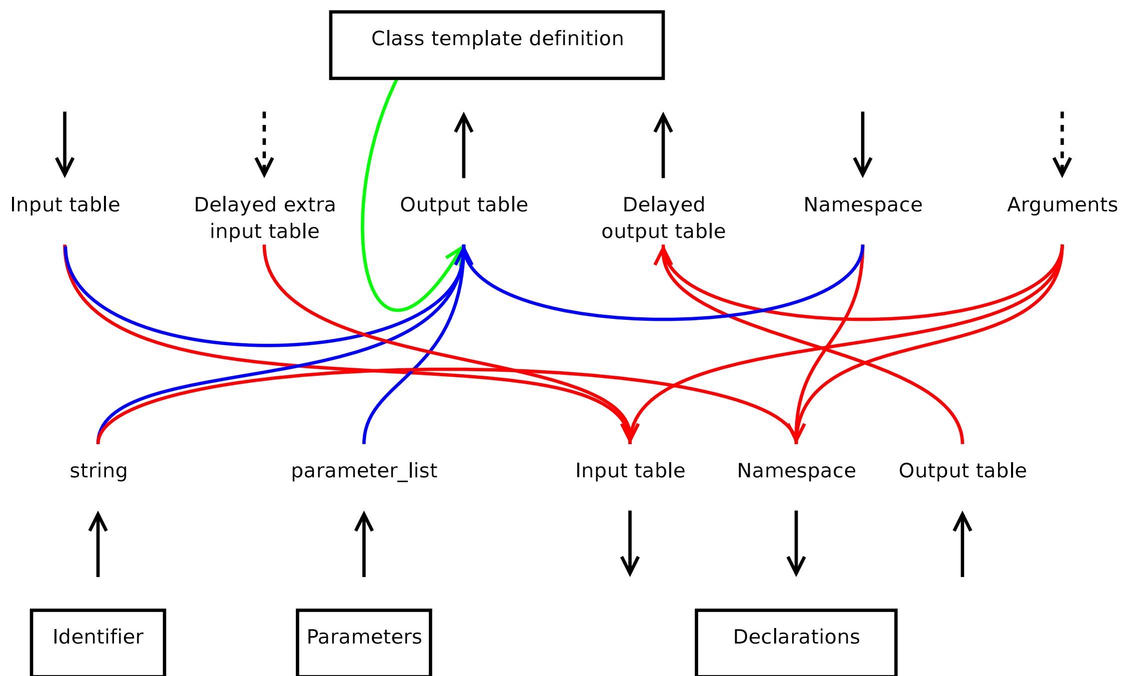

Figure 5.12: Attribute dependencies for template definition

```
"template" "<" ">" "struct" Id "<" Ty ">" "{" ds:Decl* "}" ";" -> Decl
  {attributes(disamb:
    %% Current namespace
    ds.ns           := ![Specialized(Id.string, Ty.type)|root.ns]
    Ty.ns           := !root.ns

    %% Table carrying
    Ty.in_table     := !root.in_table
    ds.in_table     := !Ty.out_table

    %% Declaration
    root.out_table  := ![([Specialized(Id.string, Ty.type)|root.ns]
                          , Class())
                        |<not(?invalid())> ds.out_table] <+ !invalid()
  )}
```

Syntax definition 5.13: Example of template specialization disambiguation

5.4 Fit with the standard

This is the general way to handle the C++ template disambiguation. But there are some little things to keep in mind to disambiguate like required by the standard. Sometimes, it is not nat-

```
Id "<" Ty ">" -> ClassName
  {attributes(disamb:
    %% Argument traversal
    Ty.ns        := !root.statement_ns
    Ty.in_table  := !root.in_table

    %%We check whether the class is already instantiated and evaluate the
    %% subtree if not
    root.new_instance := !Ty.out_table => invalid()
              <+ ( <ns_lookup> ([Specialized(Id.string , Ty.type)|root.ns],
                              <not(?invalid())> root.table) => Class() < ![] +
                <ns_lookup> ([NotSpecialized(Id.string)|root.ns],
                              <not(?invalid())> root.table) => TClass(<id>)
              ; <add-attribute>
                (<id>,
                 "disamb",
                 "extra_table",
                 <filter(\ (t,n) ->
                            (<rec x(?Ty.type; ![] <+ [id|x])> t , n) \)>
                   Ty.out_table)
              ; <add-attribute> (<id>, "disamb" , "param_type" , Ty.type)
              ; attr-eval)
            <+ !invalid()

    %%On new instance we add it in the table
    root.out_table := !root.new_instance
                    ; ( ?invalid()
                      <+ ?[]
                       ; !Ty.out_table
                      <+ <get-attribute> (<id>, "disamb",
                                          "specialized_out_table")
                       ; (?invalid() <+ <conc> (<id>, Ty.out_table))
                       )

    %% Return the new current namespace
    root.out_ns := <follow_ns> [Specialized(Id.string , Ty.type)|root.ns]

    root.type    := !Ty.type => invalid()
                    <+ ![Specialized(Id.string , Ty.type)]
  )}
```

Syntax definition 5.14: Example of template specialization disambiguation

ural because disambiguation through attribute grammars is not close to the classical approach techniques. It will require to write more code. But it will stay simple. C++ was not designed to be disambiguated by our technique, but more to fit to existing compilers.

5.4.1 Multiple delay layers in instantiation

In the described approach, something was left out for simplification purpose. But we must handle it. The instantiation is a bit more complex. When manipulating a reference to a template class, it does not need to be instantiated. But when this type is used to build an object instance, the type must be instantiated. But not completely, we only need to the declaration. Once we try to access to a member, the definition of the member is instantiated.

Program 5.15 demonstrates this. Here `A<C>::h()` is invalid because `C::t` is not a type. But it is not used, only member `g` of `A<C>` is instantiated.

```
template <typename T>
struct A {
  static void g()
  {
    int f(T::t);
    (void) f;
  }

  static void h()
  {
    int f(typename T::t);
  }
};

struct B {
  typedef int t;
};

struct C {
  static int t;
};

void f()
{
  A<C>::g();
  A<B>::h();
}
```

Program 5.15: Member instantiation

The shown approach has to be modified. Instead of delaying once until we use the type, we must delay several evaluations as described by the C++ standard.

5.4.2 Disambiguation inside of template definition

With the presented technique we are able to build the context due to template instantiation. But the disambiguation of the template definition is made on trees manipulated by attributes and not on the real tree.

Intelligent version

One way to handle it is to collect one instantiation of each template definition and then replace the tree by the instantiation. However, replacing a tree during attribute evaluation is not possible. But we can put this tree in a specific attribute that a filter will use to make the replacements.

If a template definition is instantiated several times, there is no problem we just had to verify that the disambiguation is compatible. But when there is no instantiation, we cannot disambiguate.

Standard version

In section 5.1, we have seen that the standard of the C++ had everything to disambiguate without looking for instantiation. Actually, instances are not needed because we have to require the user to write source code that conforms to the standard.

To handle the standard perfectly, a solution is to make a fake instantiation locally. This instance will be used to disambiguate. Note that we need to make fake instantiation and not traverse directly the template definition in the tree because we need to preserve the attribute as not computed since it can be computed only once.

Of course the arguments passed to the fake instantiation are specials. If they are called, the dependant code of this call is undetermined. This means that the disambiguation must behave here just like described in section 5.1 (`nested-name-specifiers` are `namespace-names`, an identifier is not a type without the `typename` keyword,...).

Hybrid version

However, the first idea must be kept. This idea make `typename` keywords optional in the template definitions. It is a nice feature because most developers forget to put it and add it finally after the first compilation tentative. It would be good to handle it so as to make transformation to automatically add these keywords.

As the standard version, there is a fake instance. But this instance has to be used only if it was not really instantiated. If an instantiation is made we just have to use (and then verify that all instances are compatible).

Chapter 6

Conclusion

Proof of concept was made. Now we have to use it and a lot of work has yet to be done.

An evaluator is working, but not efficiently. If the architecture chosen uses Stratego interpretation, it is because we certainly want to evaluate attributes during parsing so as to really merge all passes. Maybe it will help the parser since ambiguities make an explosion of the output parse forest.

Currently, a nearly complete set of tools is provided, except for a dependency loop checker.

Disambiguation filters should to be rewritten. Currently, only two filters were written. But they were simple to write despite the fact that they were representative. For example, the last filter of Transformers that checks for the kind of each type name is huge.

The type checker dependent disambiguation filters cannot be written until there is actually type checker written with attributes.

Appendix A

Mini C++

```
module TemplateStructs
exports

context-free start-symbols

  Start

sorts

  Start
  Decl
  Ty
  Var
  Id
  IDENTIFIER
  ClassName
  TypedefName
  EnumName
  Enumerator
  EnumeratorName
  Namespace

context-free syntax

  ds:Decl* -> Start
    {attributes(disamb:
      ds.table := ![]

      ds.ns      := ![]

      root.instances      := !ds.instances

      ds.down_instances :=
          !root.instances
```

```
      ; map(\ (t, _) -> t \)
      ; uniq
      ; map({n, first:
            ?n
          ; <filter(\ (n, tree) -> tree \)> root.instances
          ; ?[first|_]
          ; map(rec x(
                      is-int
                  <+ \ attr-appl(_, c, a, _, _)
                        -> sign(<map(x)> c,
                               <filter(?(_, "ok", <id>))> a) \
                  <+ \ amb(l) -> amb(<map(x)> l) \
                      )
              )
          ; uniq
          ; ( ?[_]
           <+ <debug(!"Warning␣ambiguous␣use␣of␣template:␣")> n)
          ; !(n, first)
          })
  )}

ds:Decl+ -> Decl*
  {attributes(disamb:
    ds.ns           := !root.ns
    ds.table        := !root.table
    root.out_table  := !ds.out_table

    root.instances     := !ds.instances
    ds.down_instances  := !root.down_instances
  )}
          -> Decl*
  {attributes(disamb:
    root.out_table  := !root.table
    root.instances  := ![]
  )}

l:Decl+ r:Decl+ -> Decl+
  {attributes(disamb:
    l.ns              := !root.ns
    r.ns              := !root.ns

    l.table           := !root.table
    root.temp_table   := !l.out_table

    r.table           := !root.temp_table
    root.out_table    := !r.out_table

    root.instances := <conc> (<not(?invalid())> l.instances,
                              <not(?invalid())> r.instances) <+ !invalid()
```

```
      l.down_instances     := !root.down_instances
      r.down_instances     := !root.down_instances
    )}

  Decl                 -> Decl+
    {attributes(disamb:
      Decl.ns           := !root.ns

      Decl.table        := !root.table
      root.out_table  := !Decl.out_table

      root.instances          := !Decl.instances
      Decl.down_instances  := !root.down_instances
    )}

  "template" "<" "typename" tn:Id ">" "struct" sn:Id "{" ds:Decl* "}" ";"
    -> Decl
    {attributes(disamb:
      root.param_type    := extern
      root.extra_table   := extern

      ds.ns      := ![Specialized(sn.string, root.param_type)|root.ns]
      ds.table := <conc> (<map(\ (l, t)
                               -> (<conc> (l, [Type(tn.string)]), t) \)>
                               root.extra_table,
                             root.table)
      root.out_table  := ![([NotSpecialized(sn.string)|root.ns]
                            , TClass(<id>))
                             |<not(?invalid())> root.table] <+ !invalid()

      root.specialized_out_table  := !ds.out_table => invalid()
                                     <+ <diff> (ds.out_table, ds.table)

      root.instances  := ![] then we evaluate.
      root.specialized_instances  := !ds.instances

      root.replace := <lookup> ([NotSpecialized(sn.string)|root.ns],
                                root.down_instances)
      ds.down_instances  := extern
    )}

  "template" "<" ">" "struct" Id "<" Ty ">" "{" ds:Decl* "}" ";" -> Decl
    {attributes(disamb:
      ds.ns             := ![Specialized(Id.string, Ty.type)|root.ns]
      ds.table          := !Ty.out_table

      root.out_table  := ![([Specialized(Id.string, Ty.type)|root.ns]
                              , Class())
                             |<not(?invalid())> ds.out_table] <+ !invalid()
```

```
    Ty.table          := !root.table
    Ty.ns             := ![]

    root.instances := <conc> (<not(?invalid())> Ty.instances,
                              <not(?invalid())> ds.instances) <+ !invalid()
    Ty.down_instances := !root.down_instances
    ds.down_instances := !root.down_instances
  )}

"struct" Id "{" ds:Decl* "}" ";" -> Decl
  {attributes(disamb:
    ds.ns             := ![Type(Id.string)|root.ns]
    ds.table          := !root.table
    root.out_table  := ![([Type(Id.string)|root.ns], Class())
                          |<not(?invalid())> ds.out_table] <+ !invalid()

    root.instances    := !ds.instances
    ds.down_instances := !root.down_instances
  )}

"enum" Id "{" es:{Enumerator ","}* "}" ";" -> Decl
  {attributes(disamb:
    es.ns             := ![Type(Id.string)|root.ns]
    es.table          := !root.table
    root.out_table  := ![([Type(Id.string)|root.ns], Enum())
                          |<not(?invalid())> es.out_table] <+ !invalid()
    root.instances  := ![]
  )}

es:{Enumerator ","}+ -> {Enumerator ","}*
  {attributes(disamb:
    es.ns             := !root.ns
    es.table          := !root.table
    root.out_table  := !es.out_table
  )}

                    -> {Enumerator ","}*
  {attributes(disamb:
    root.out_table  := !root.out_table
  )}

l:{Enumerator ","}+ "," r:{Enumerator ","}+ -> {Enumerator ","}+
  {left,
   attributes(disamb:
    l.ns              := !root.ns
    r.ns              := !root.ns

    l.table            := !root.table
```

```
    root.temp_table := !l.out_table
    r.table         := !root.temp_table
    root.out_table  := !r.out_table
  )}

Enumerator -> {Enumerator ","}+
  {attributes(disamb:
    Enumerator.ns     := !root.ns
    Enumerator.table  := !root.table
    root.out_table    := !Enumerator.out_table
  )}

Id -> Enumerator
  {attributes(disamb:
    root.out_table := ![([Enumerator(Id.string)|root.ns], Enumerator())
                       |root.table]
  )}

Id -> Var
  {attributes(disamb:
    root.ok := <lookup> ([Var(Id.string)|root.ns],
                        <not(?invalid())> root.table) => Var()
               < !1 + !0

    root.instances := !root.ok => 1 < ![] + !invalid()
  )}

Var -> Expr
  {attributes(disamb:
    Var.ns          := ![]
    Var.table       := !root.table
    root.out_table  := !root.table

    root.instances       := !Var.instances
    Var.down_instances   := !root.down_instances
  )}

Id -> EnumeratorName
  {attributes(disamb:
    root.ok := <lookup> ([Var(Id.string)|root.ns],
                        <not(?invalid())> root.table) => Enumerator()
               < !1 + !0

    root.instances := !root.ok => 1 < ![] + !invalid()
  )}

EnumeratorName -> Expr
  {attributes(disamb:
    EnumeratorName.ns     := !root.ns
```

```
      EnumeratorName.table  := !root.table
      root.out_table        := !root.table

      root.instances                  := !EnumeratorName.instances
      EnumeratorName.down_instances  := !root.down_instances
    )}

  "static" Ty Id ";" -> Decl
    {attributes(disamb:
      root.out_table := ![([Var(Id.string)|root.ns], Var())
                          |<not(?invalid())> Ty.out_table] <+ !invalid()
      Ty.table       := !root.table
      Ty.ns          := ![]

      root.instances     := !Ty.instances
      Ty.down_instances  := !root.down_instances
    )}

  Ty ";" -> Decl
    {attributes(disamb:
      Ty.ns          := ![]
      Ty.table       := !root.table
      root.out_table := !root.table

      root.instances     := !Ty.instances
      Ty.down_instances  := !root.down_instances
    )}

  Expr ";" -> Decl
    {attributes(disamb:
      Expr.ns        := ![]
      Expr.table     := !root.table
      root.out_table := !root.table

      root.instances       := !Expr.instances
      Expr.down_instances  := !root.down_instances
    )}

%% Id -> Ty
  ClassName -> Ty
    {attributes(disamb:
       ClassName.table  := !root.table
       root.out_table   := !ClassName.out_table
       ClassName.ns     := !root.ns
       root.out_ns      := !ClassName.out_ns
       root.type        := !ClassName.type

       root.instances             := !ClassName.instances
       ClassName.down_instances  := !root.down_instances
```

```
  )}

EnumName -> Ty
  {attributes(disamb:
     EnumName.table  := !root.table
     root.out_table  := !EnumName.out_table
     EnumName.ns     := !root.ns
     root.out_ns     := !EnumName.out_ns
     root.type       := !EnumName.type
     root.instances  := !EnumName.instances
  )}

TypedefName -> Ty
  {attributes(disamb:
     TypedefName.table  := !root.table
     root.out_table     := !TypedefName.out_table
     TypedefName.ns     := !root.ns
     root.out_ns        := !TypedefName.out_ns
     root.type          := !TypedefName.type
     root.instances     := !TypedefName.instances
  )}

ClassName -> Namespace
  {attributes(disamb:
     ClassName.table  := !root.table
     root.out_table   := !ClassName.out_table
     ClassName.ns     := !root.ns
     root.out_ns      := !ClassName.out_ns
     root.type        := !ClassName.type

     root.instances              := !ClassName.instances
     ClassName.down_instances  := !root.down_instances
  )}

TypedefName -> Namespace
  {attributes(disamb:
     TypedefName.table  := !root.table
     root.out_table     := !TypedefName.out_table
     TypedefName.ns     := !root.ns
     root.out_ns        := !TypedefName.out_ns
     root.type          := !TypedefName.type
     root.instances     := !TypedefName.instances
  )}

Id -> ClassName
  {attributes(disamb:
    root.ok := <lookup> ([Type(Id.string)|root.ns],
                         <not(?invalid())> root.table)
               => Class() < !1 + !0
```

```
    root.out_table := !root.ok => 1 < !root.table + !invalid()
    root.out_ns     := !root.ok => 1 < ![Type(Id.string)|root.ns]
                 + !invalid()
    root.type       := !root.ok => 1 < ![Type(Id.string)] + !invalid()

    root.instances := !root.ok => 1 < ![] + !invalid()
  )}

Id -> EnumName
  {attributes(disamb:
    root.ok := <lookup> ([Type(Id.string)|root.ns],
                         <not(?invalid())> root.table)
               => Enum() < !1 + !0

    root.out_table := !root.ok => 1 < !root.table + !invalid()
    root.out_ns     := !root.ok => 1 < ![Type(Id.string)|root.ns]
                 + !invalid()
    root.type       := !root.ok => 1 < ![Type(Id.string)] + !invalid()

    root.instances := !root.ok => 1 < ![] + !invalid()
  )}

Id -> TypedefName
  {attributes(disamb:
    root.ok := <lookup> ([Type(Id.string)|root.ns],
                         <not(?invalid())> root.table)
               => Typedef() < !1 + !0

    root.out_table := !root.ok => 1 < !root.table + !invalid()
    root.out_ns     := !root.ok => 1 < ![Type(Id.string)|root.ns]
                 + !invalid()
    root.type       := !root.ok => 1 < ![Type(Id.string)] + !invalid()

    root.instances := !root.ok => 1 < ![] + !invalid()
  )}

Id "<" Ty ">" -> ClassName
  {attributes(disamb:
    root.out_table := !root.new_instance
                    ; ( ?invalid()
                     <+ ?[]
                      ; !Ty.out_table
                     <+ <get-attribute> (<id>, "disamb",
                                         "specialized_out_table")
                      ; (?invalid() <+ <conc> (<id>, Ty.out_table))
                      )

    root.out_ns := ![Specialized(Id.string, Ty.type)|root.ns]
```

```
    Ty.ns          := ![]
    Ty.table       := !root.table

    root.type      := !Ty.type => invalid()
                      <+ ![Specialized(Id.string, Ty.type)]

    Ty.down_instances := !root.down_instances

    root.new_instance := !Ty.out_table => invalid()
                <+ ( <lookup> ([Specialized(Id.string, Ty.type)|root.ns],
                           <not(?invalid())> root.table) => Class() < ![] +
                  <lookup> ([NotSpecialized(Id.string)|root.ns],
                           <not(?invalid())> root.table) => TClass(<id>)
                ; <add-attribute>
                  (<id>,
                   "disamb",
                   "extra_table",
                   <filter(\ (t,n) ->
                            (<rec x(?Ty.type; ![] <+ [id|x])> t, n) \)>
                     Ty.out_table)
                ; <add-attribute> (<id>, "disamb", "param_type", Ty.type)
                ; attr-eval)
              <+ !invalid()

    root.instances := !root.new_instance
                      ; ( ?invalid()
                        <+ ( ?[]
                           < !Ty.instances
                           + ![([NotSpecialized(Id.string)|root.ns], <id>)|
                               <conc>
                                 (<get-attribute
                                 ; not(?invalid())> (<id>, "disamb",
                                                     "specialized_instances"),
                                  <not(?invalid())> Ty.instances
                                 )
                              ]
                           )
                         )
                      <+ !invalid()
  )}

l:Namespace "::" r:Namespace -> Namespace
  {left,
   attributes(disamb:
    l.table := !root.table
    root.temp_table := !l.out_table
    r.table          := !root.temp_table
    root.out_table   := !r.out_table
```

```
    l.ns := !root.ns
    root.temp_ns := !l.out_ns
    r.ns         := !root.temp_ns
    root.out_ns  := !r.out_ns

    root.type     := <conc> (<not(?invalid())>r.type,
                             <not(?invalid())>l.type) <+ !invalid()

    root.instances := <conc> (<not(?invalid())> l.instances,
                              <not(?invalid())> r.instances) <+ !invalid()

    l.down_instances := !root.down_instances
    r.down_instances          := !root.down_instances
   )}

 Namespace "::" Ty -> Ty
   {left,
    attributes(disamb:
     Namespace.table := !root.table
     root.temp_table := !Namespace.out_table
     Ty.table        := !root.temp_table
     root.out_table  := !Ty.out_table

     Namespace.ns := !root.ns
     root.temp_ns := !Namespace.out_ns
     Ty.ns        := !root.temp_ns

     root.type     := <conc> (<not(?invalid())> Ty.type,
                             <not(?invalid())> Namespace.type) <+ !invalid()

     root.instances := <conc> (<not(?invalid())> Namespace.instances,
                              <not(?invalid())> Ty.instances) <+ !invalid()

     Namespace.down_instances := !root.down_instances
     Ty.down_instances         := !root.down_instances
   )}

 Namespace "::" Var -> Var
   {left,
    attributes(disamb:
     Namespace.table := !root.table
     root.out_table  := !Namespace.out_table

     Namespace.ns := !root.ns
     root.temp_ns := !Namespace.out_ns
     Var.ns       := !root.temp_ns
     Var.table    := !Namespace.out_table
```

```
    root.instances := <conc> (<not(?invalid())> Namespace.instances,
                              <not(?invalid())> Var.instances) <+ !invalid()

    Namespace.down_instances := !root.down_instances
    Var.down_instances       := !root.down_instances
  )}

Namespace "::" EnumeratorName -> EnumeratorName
  {left,
   attributes(disamb:
    Namespace.table := !root.table
    root.out_table  := !Namespace.out_table

    Namespace.ns := !root.ns
    root.temp_ns := !Namespace.out_ns
    EnumeratorName.ns       := !root.temp_ns
    EnumeratorName.table    := !Namespace.out_table

    root.instances := <conc> (<not(?invalid())> Namespace.instances,
                              <not(?invalid())> EnumeratorName.instances)
                      <+ !invalid()

    Namespace.down_instances      := !root.down_instances
    EnumeratorName.down_instances := !root.down_instances
  )}

"typedef" Ty Id ";" -> Decl
  {attributes(disamb:
    Ty.ns           := ![]
    Ty.table        := !root.table
    root.out_table := <filter(\ (t,n) -> (<rec x(?Ty.type; ![]
                                                <+ [id|x])> t, n) \)>
                          Ty.out_table
                     ; one(\ ([], n) -> ([], Typedef()) \)
                     ; map(\ (l, n) ->
                          (<conc> (l, [Type(Id.string)|root.ns]), n) \)
                     ; <conc> (<id>, Ty.out_table)

    root.instances     := !Ty.instances
    Ty.down_instances := !root.down_instances
  )}


IDENTIFIER -> Id
  {attributes(disamb:
    root.string :=
          rec x(![<is-int>] <+ ?attr-appl(_, <map(x); concat>, _, _, _))
        ; implode-string
  )}
```

```
lexical syntax
  [a-zA-Z\_][a-zA-Z0-9\_]* -> IDENTIFIER
  [\ \r\t\n] -> LAYOUT
lexical restrictions
  Id -/- [a-zA-Z0-9\_]
lexical restrictions
  LAYOUT? -/- [\ \r\t\n]
```

Appendix B

Bibliography

Anisko, R., David, V., and Vasseur, C. (2003). Transformers: a c++ program transformation framework. Technical report, LRDE.

Bravenboer, M. (2003). Stratego shell.

C++ standard (2003). Programming languages - c++. ISO/IEC 14882.

Dick Grune, Henri E. Bal, C. J. J. and G.Langendoen, K. (2000). *Modern Compiler Design*. John, Wiley & Sons. The most complete source. Moreover all is not in it.

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.

Stratego (2000). Stratego/xt.

Swierstra, S. D., Alcocer, P. R. A., and Sariava, J. (1998). Designing and implementing combinator languages. In *Advanced Functional Programming*, pages 150–206. An application of AGS.

UU AG (1999). Uu attribute grammar system. Attribute grammar framework in Haskell.

Visser, E. (2001). Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In Middeldorp, A., editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag.

Index

ambiguities, 9
    kinds, 9
    namespaces, 9, 21–22
    template parameters, 9, 19–21
AsFix, 13
attributes, 11

C++ standard, 9
check, 12
    cycles, 12
    definitions, 12, 16

evaluator, 11–12, 16

problems, 8–9, 18

sdf-attribute, 13–16, 36
SGLR, 8, 13, 18
Stratego, 8, 9, 13

Transformers, 8, 36

XT, 8, 13
    SGLR, 8, 13, 18
    Stratego, 8, 9, 13