Many software engineering problems, such as refactoring or optimisation, can be efficiently solved by using source-to-source program transformation technologies. Moreover, in the specific case of the C++ language, automatic program transformations can be used as an attempt to bridge the gap between a classic programming style, and the intensive meta-programming techniques involved in generative libraries.

In this report, we share our experience in the development of a C++ program transformation framework, ranging from our selection of meta-tools and the architecture of our system, to issues relevant to the bad syntactic and semantic properties of the language itself. Also, some new perspectives on active libraries are discussed.

Keywords
C++ language, Parsing, Program transformation
# Contents

1 Introduction .......................................................... 6
1.1 Why program transformation? ........................................... 6
1.2 Applying program transformation to generative libraries ................. 6
1.2.1 The libraries and the language ....................................... 7
1.2.2 Drawbacks of generic programming ................................... 7
1.2.3 A solution .......................................................... 7

2 Tools and architecture of the framework .................................. 9
2.1 Meta-tools ............................................................. 9
2.1.1 Parsing with Sglr .................................................. 9
2.1.2 Rewriting with Stratego ............................................ 10
2.2 Architecture .......................................................... 10
2.2.1 Grammar and derived products ..................................... 10
2.2.2 Processing chain .................................................. 12

3 Ambiguous parsing and parse forest filtering .............................. 15
3.1 Overview .............................................................. 15
3.2 Local and specific filters ............................................... 17
3.2.1 Generic ambiguity removal ......................................... 18
3.2.2 Namespace definitions ............................................. 19
3.2.3 Post-processing declarations ..................................... 20
3.3 Semantic analysis ...................................................... 23
3.3.1 Classifying declarations ........................................... 24
3.3.2 Reading partially ambiguous declarations ......................... 24

4 Conclusion and further work ............................................. 26
4.1 Limitations ........................................................... 26
4.2 Further work .......................................................... 26
List of Figures

2.1 Definitions in C++ .................................................. 11
2.2 C++ grammar processing ........................................... 12
2.3 Removing optional literals - source .............................. 12
2.4 Removing optional literals - result ............................... 12
2.5 Annotating with constructors - source .......................... 13
2.6 Annotating with constructors - result ............................ 13
2.7 Signature generation - source ..................................... 13
2.8 Signature generation - result ...................................... 13
2.9 C++ program processing ............................................ 14

3.1 A parse forest ......................................................... 15
3.2 A post-processed parse forest ...................................... 16
3.3 Filtering chain for parse forests ................................. 17
3.4 afcpp-resolve .......................................................... 18
3.5 Production rules for namespace definitions ...................... 19
3.6 An ambiguous declarator identifier ............................... 22
List of Programs

3.1 A simple declaration .................................................. 16
3.2 afcpp-resolve ........................................................... 18
3.3 Namespace definition versus namespace extension ............... 19
3.4 Grouping in declarations .............................................. 21
3.5 Another simple declaration .......................................... 21
3.6 Qualified names in lists of specifiers ............................... 23
3.7 A class declaration ..................................................... 24
List of Algorithms

3.1 afcpp-namespace ................................................. 19
3.2 afcpp-declaration ............................................... 21
3.3 afcpp-declarator ............................................... 22
3.4 afcpp-specifier .................................................. 23
3.5 afcpp-disambiguate ............................................. 25
Chapter 1

Introduction

This report is mainly a case study in program transformation. It discusses the development of a transformation system dedicated to the C++ language.

More specifically, we focus on the meta-tools being used for the implementation of our framework, the global architecture of our system with respect to these implementation tools, and describe the issues encountered when dealing with C++.

1.1 Why program transformation?

Program transformation is indeed a very general term, which can cover both many different applications and many different technologies. However, it usually denotes transformations based on the structure of the programs being manipulated; traditionally performed with tree rewriting systems.

The very first consequence of using tree rewriting techniques is, obviously, the need for tools suitable to the implementation of the usual processing steps on trees:
- Syntactic analysis.
- Rewriting.
- Unparsing or pretty-printing.

The second consequence of using tree rewriting systems, instead of weaker techniques, such as, for example, mere text replacement, is simply the ability to cover a broad range of applications:
- Program refactoring and renovation.
- Program documentation and instrumentation.
- Translation and compilation.
- High-level optimization, such as partial evaluation, for example.

This list is far from being complete. For a detailed taxonomy of program transformation, see Visser and Deursen (2000).

1.2 Applying program transformation to generative libraries

Above, we have seen an overview of the traditional motivations for using program transformation, but our interest in these techniques is more specific.

Actually, the applications of program transformation we are targeting at are closely related to the development of two generative libraries, Olena (dedicated to image processing) and Vaucanson (dedicated to finite state machines).
1.2.1 The libraries and the language

These libraries have very similar goals. They mainly aim at providing an high level of genericity (the ability to process inputs of very different kinds), while keeping at the same time an high level of performance (no abstraction penalty, no additional run-time cost).

In practice, this can be achieved in C++ with the generic programming paradigm, by using so-called meta-programming techniques, which rely on compile-time computations, and the fact that C++ actually became a two-level language after the introduction of parametric polymorphism, and template constructions.

A discussion of C++ as a two-level language and the relationship with partial evaluation can be found in Veldhuizen (1999). The generic programming paradigm and its application in the Olena image processing library are discussed in Géraud (2002) and Darbon et al. (2002).

1.2.2 Drawbacks of generic programming

Unfortunately, although these programming techniques enable us to write both generic and efficient libraries, they suffer from a number of major issues:

- Since meta-programming involves compile-time computations and forbids separate compilation, the compilation process becomes extremely heavy.
- Worse, generic programming tends to make programs very complex and difficult to write, at least from the library implementor point of view.
- Of course, generative libraries are as well difficult to read, which makes the debugging and maintenance a tedious task.

1.2.3 A solution

Simplifying generic programming

Obviously, this is were program transformation techniques are needed.

In order to improve development in the generic programming paradigm, we intend to automate the process of deriving active libraries, that make use of intensive meta-programming techniques, from libraries written in a much simpler way, akin to the usual C++ programming style; in other words, classic C++, possibly equipped with some syntactic extensions designed to capture some concepts that do exist only in generic programming.

The constraints

Of course, the need for applying any kind of structure-based transformation to C++ programs gave birth to a project of itself: the development of a transformation framework dedicated to the C++ language, whose first sketch is presented in this report.

Keep in mind, though, that this project was initiated with some strong constraints in mind, most noticeably concerning the grammar being used for the C++ language.

Our grammar was first extracted from the grammar given in the C++ language standard, and we tried to remain as close as possible to this original grammar, even though it is far from being perfect. There are several motivations for this guideline:

- At first, it makes our grammar very close to the reference grammar. This is important when considering our transformation system as a stand-alone project: it seems to be a quite reasonable claim for people working with C++ to be able to manipulate a grammar being nearly the standardized grammar.
• While the standard grammar is not flawless\textsuperscript{1}, it is probably the simplest possible grammar for C++. This is a good property in the context of program transformation, for the grammar determines the shape of parse and abstract syntax trees; the more the grammar is complex, the more the trees are complex, and the transformations longer to specify.

\textsuperscript{1}And cannot be. As it is explained further on in this report, this grammar can only be ambiguous.
Chapter 2

Tools and architecture of the framework

This chapter briefly describes the meta-tools that have been chosen for the implementation of our transformation system, and explains the motivations behind these choices. The global architecture of our system is then presented, and we show how it is integrated with this collection of meta-tools.

2.1 Meta-tools

Most of the tools and technologies being used for the implementation of our transformation system are imported from the following projects or collections of tools:

- The ASF+SDF Meta-Environment, developed at the Centrum voor Wiskunde en Informatica under the Generic Language Technology project (Brand et al., 2001).

- The Stratego (Visser, 2001) language for specification of program transformations. The Stratego compiler is currently under development at the Utrecht University. The language and compiler were first prototyped in the Pacific Software Research Center, at the Oregon Graduate Institute.

- The generic pretty-printer GPP (Jonge, 2000).

- XT (Jonge et al., 2001; Jonge and Visser, 2001b), a collection of program transformation tools. This bundle includes among other things the Sglr parser, the Stratego compiler, and GPP.

2.1.1 Parsing with Sglr

One of the essential components used in our transformation system is the generic Sglr parser. This powerful tool implements scanner-less generalised LR parsing, and provides a large amount of advantages over traditional parsing techniques:

- There are no restrictions on the class of grammars that can be handled: all context-free grammars can be addressed. In practice, this implies that there is absolutely no need to massage and obfuscate a grammar before its use.

- Generalised parsing is not deterministic. Thus, when dealing with an ambiguous grammar, the parser can build a parse forest rather than a single tree. This is especially useful when working with a language such as C++, that suffers from non-syntactic ambiguities.
• Context-free grammars are closed under union, unlike subclasses such as LALR. The first consequence of this is the ability, when working with Sglr and the SDF grammar formalism, to define modular grammars; the readability and maintainability of the grammars being developed is significantly increased.

• This grammar modularity is also of interest in the specific case of our work, because we aim, among other things, at introducing various extensions to C++, possibly giving birth to different flavours of the language. In this context, we cannot afford the use of several one-chunk grammars that would make the maintenance of the shared core language unmanageable.

A more detailed description of Sglr can be found in Brand et al. (2002).

2.1.2 Rewriting with Stratego

Most software components of our system are specified in Stratego, a language dedicated to program transformation, based on rewriting strategies. From these specifications, the Stratego compiler produces C code that is used to build the stand-alone programs that compose our processing chains.

In the specific case of our C++ transformation system, the following features are of interest in Stratego:

• Stratego primarily supports specifications of transformations on abstract syntax trees, unlike the ASF+SDF Meta-Environment, that focuses on concrete syntax. This is of prime importance when working with a language whose concrete syntax is ambiguous.

• Transformations can also be defined directly on parse trees in AsFix format. This can be very helpful to address informations that cannot be represented in abstract syntax trees of the object language, such as parsing ambiguities.

• The presence of rewriting strategies, used to define how and when rewrite rules should be applied, contributes to augment significantly the modularity and reusability of specifications. The strategic programming paradigm is discussed in Lämmel et al. (2002).

• Last, Stratego is provided with a very complete library of rules and strategies that implement generic traversals, standard data types, or various system interfaces (Visser, 2000).

2.2 Architecture

With respect to many of its aspects, our transformation system is extremely classic, and similar to many other projects that make use of the meta-tools described in section 2.1.

2.2.1 Grammar and derived products

Due to the nature of the implementation tools, the grammar plays a central role in our architecture. Of course, it is used to generate the parse tables for the syntactic analysis stage, but also serves to the communication of parse trees and syntax trees between the various software components, by acting as a contract (Jonge and Visser, 2001a). Last, the grammar is a basis for the generation of a pretty-printing table.

Thus, several processing stages in our system are dedicated to transformations on the C++ grammar.
Grammar

The base grammar we have developed is based on the extended BNF specification of the C++
ISO/IEC international standard (iso, 1998). This work was mostly a translation to SDF: excepted
for some minor changes and corrections, the standard grammar did require no massaging. In-
cluding both lexical and context-free syntax, our grammar is approximately 520 rules big.
Unfortunately, this grammar is far from being perfect:

- Since C++ is a context dependent language, the only possible way of making its grammar fit
into a context-free specification is to produce an ambiguous grammar. This is the grammar
we refer to as being our grammar for C++ language; actually, it does not define strictly the
syntax of C++, but a super-set of the language.

Of course, this is a major issue in our transformation system: a large effort must be devoted
to a semantic analysis stage, whose purpose is the removal of parsing ambiguities.

- In this grammar, some rules do not constrain enough the input texts. This is the case in
particular for definitions, which are all handled by a single rule, shown in figure 2.1.

Some rules of this kind have been left unchanged in the grammar, to remain as close as
possible to the reference grammar of the standard, but also because these very simple rules
define each time a large amount of correct constructions; enumerating only all well-formed
constructions would have certainly caused a blowup of the number of rules in the grammar.

Figure 2.1 Definitions in C++

```
DeclSpecifierSeq: InitDeclaratorList? ; → SimpleDeclaration
```

Because of such rules, many ill-formed programs are accepted at parsing-time. As a con-
sequence, there is a need in our framework to analyse the trees returned by the parser not
only to disambiguate them, but also to reject invalid inputs.

Grammar processing

As described above, this grammar is processed by several fully automated tools before it is ac-
tually used for program transformations (a summary of this processing chain is given in figure
2.2):

- Some minor corrections are applied to the original grammar. A tool named sdf-option
introduces intermediate non-terminals to remove optional literals. Optional literals are a
very convenient construct, as shown by figure 2.3, but they are troublesome in abstract
syntax trees. The grammar excerpt from figure 2.3 is transformed in the rules shown in
figure 2.4.

- The original grammar is annotated with constructors, using sdf-cons. This tool associates
a constructor to each production rule; this constructor is a label used during implosions of
parse trees into abstract syntax trees. The grammar chunk of figure 2.5 is transformed to
the rules shown in figure 2.6.

- The annotated grammar is used to produce the parsing table, with the help of the generator
sdf2table.

- The annotated grammar is used to produce a Stratego signature using sdf2sig. This sig-
nature is then used to define transformations on abstract syntax trees. Figures 2.7 and 2.8
give a small example of signature generation.

- A pretty-printing table is generated from the annotated grammar using ppgen.
2.2 Architecture

When all the informations derived from the grammar have been successfully generated, the remaining software components of our transformation system are built from their Stratego specifications.

Unlike many transformation software based on Sglr and Stratego, the core of our system is not
based solely on our grammar of the C++ language. Where a typical processing chain branches the various rewriting components after the syntactic analysis stage, we introduce a post-processing step to assist the parser.

The need for this post-processing stage is due to the very nature of the grammar being used for the C++ language: as mentioned in section 2.2.1, this grammar actually defines a super-set of C++ to satisfy our context-freedom requirement. Therefore, we end up with a syntactic analyzer that is capable of reading an input text in several different fashions; upon success, it returns a parse forest rather than a single tree.

Of course, the purpose of these post-processing tools is to perform various analyses on the parse forest to filter it, and reduce it to one correct tree that can be finally shipped to the transformation components.

As a consequence, the processing of a C++ program includes the following steps:

- Parsing with sglr and the table generated from the grammar.
- Reduction of the parse forest by sequential application of several disambiguation tools. This step is discussed in depth in chapter 3.
- Implosion of the resulting parse tree to an abstract syntax tree. This is done by invoking implode-asfix.
- Transformation of the abstract syntax tree.
• Pretty-printing with \texttt{ast2abox} and a box back-end (such as \texttt{abox2text}, \texttt{abox2html}, or \texttt{abox2latex}).

For some applications, imploding the parse tree is to be avoided\textsuperscript{1}. In such cases, an alternative processing chain is used, where transformations work directly on parse trees, and the result is unparsed, rather than pretty-printed. The diagram shown in figure 2.9 is a summary of the possible processing chains.

\textbf{Figure 2.9} C++ program processing

\begin{itemize}
\item \texttt{Parsing Table}
\item \texttt{sglr}
\item \texttt{Raw Parse Forest}
\item \texttt{Disambiguate}
\item \texttt{Single Parse Tree}
\item \texttt{implode-asfix}
\item \texttt{Abstract Syntax Tree}
\item \texttt{Transformation}
\item \texttt{Resulting Parse Tree}
\item \texttt{Unparsing}
\item \texttt{Transformed C++ Text}
\item \texttt{ast2abox}
\item \texttt{abox2text}
\item \texttt{abox2html}
\item \texttt{abox2latex}
\end{itemize}

\textbf{Fixme:} Give some details about transformations?\footnote{Working on parse trees can be needed in some applications, for example to keep track of the layout and comments.}
Chapter 3

Ambiguous parsing and parse forest filtering

This chapter focuses more specifically on selected parts of our approach to program transformation of C++. In particular, we show how our system relies on a non-deterministic parser assisted by a post-processing stage to deal properly with the C++ language, without sacrificing our grammar.

3.1 Overview

As stated in chapter 2, the architecture of our system is based on having a simple grammar of the language, assisted by a post-processing stage supposed to correct its deficiencies.

This post-processing step is actually a collection of filters specified in Stratego, like any other transformation component. The major difference is that these filters work at a lower level, that is, directly on parse trees rather than on abstract syntax trees, to be able to see and handle parsing ambiguities.

An example of an input parse forest is given in figure 3.1, where ambiguities are depicted by diamond-shaped nodes. This sample tree shows how, from an ambiguous input text (in the
present instance, the program 3.1), the Sglr parser is able to produce a very concisely encoded parse forest\(^1\).

**Program 3.1** A simple declaration

```c
typedef int foo; // Removed from the parse tree for clarity.
foo bar;
```

The post-processing is performed on a forest of this kind, and is expected to completely remove all ambiguous nodes. The resulting tree is shown in figure 3.2. As expected, on every diamond-shaped node, a choice has been made, and only one branch has been kept; the forest is reduced to a single tree, which represents the only correct parsing of the input text.

**Figure 3.2** A post-processed parse forest

The filtering stage in itself is a multi-stage process, composed of many more-or-less complex transformation components, applied in the following order:

1. **afc++-namespace**: Namespace definitions need to be filtered, among other things because our grammar makes a distinction between the first definition of a namespace, and the extension of an already defined namespace.

2. **afc++-declaration**: As explained in section 2.2.1, declarations are a major source of ambiguities. This program is one of the several components that contribute to filter declarations.

3. **afc++-resolve**: Some of our filters do not directly remove ambiguities, but cut branches they consider as being not valid. At some point, when an ambiguity node has only one child left, it can be removed. This generic ambiguity resolution is performed by **afc++-resolve**.

4. **afc++-typedef**: This component performs some processing on type declarations.

5. **afc++-resolve**: Applied a second time.

\(^1\)A parse forest of this kind, while its size remains fairly reasonable, stores an exponential number of parse trees.
6. afcpp-declarator: This filter processes declarations.

7. afcpp-specifier: This filter also processes declarations.

8. afcpp-disambiguate: This is the last and most complex filter. When a parse forest has passed through the previous components, it has been reduced enough to be suitable for this large semantic analysis stage: afcpp-disambiguate walks the whole program to determine the kind of each symbol; with this knowledge, a second traversal finishes the reduction of the parse forest.

A summary of this processing chain is given in figure 3.3. More detailed descriptions of the components are given further in the present chapter.

3.2 Local and specific filters

In this section, we discuss the simple filters that are applied to parse forests before the final analysis stage. In the processing chain described above, they range from afcpp-namespace to afcpp-specifier.
3.2 Local and specific filters

3.2.1 Generic ambiguity removal

The simplest of our filters is afcpp-resolve. As described above, it is a generic filter which removes ambiguities that have already been reduced enough. An example of this process is given in figure 3.4.

Since this filter is trivial enough, we will comment here its Stratego specification as an example. It is given in program 3.2.

As explained in section 2.1.2, the specification given above is composed of rules and strategies:

- Rules describe the actual transformations to be performed. In this case, we simply state that an ambiguous node to which only one subtree is attached, is transformed into this unique son.
- On the other hand, strategies are used to describe when and how the rewriting rules should be applied. In the case of afcpp-namespace, the processing we need is very simple: our rule is applied during a top-down traversal of the input tree. Also, our rule is wrapped in a try strategy: each node for which our rule fails to match is left unchanged, which is precisely the expected behaviour. Last, the iowrap strategy provides a very convenient interface with the system: it reads a tree on standard input, applies the strategy passed in parameter, and writes the resulting tree on standard output.

This example shows the benefits of using Stratego. With a very concise specification, we get a stand-alone software component that implements the wanted transformation: all ambiguities with a single branch left are removed.

Further on in this report, we will no more enter into such implementation details, since the specifications will tend to become a lot more complex.
3.2.2 Namespace definitions

In the processing chain described earlier in this chapter, the very first filter applied to parse forests relates to namespace definitions.

The need for this post-processing is quite simple: in C++, namespace definitions do not have to be unique; actually, a given namespace can be defined at some place, but extended later with more members (a very basic example can be found in program 3.3).

```cpp
namespace foo { int a ; } // First definition of namespace foo.
namespace bar { int a ; } // First definition of namespace bar.
namespace foo { int b ; } // Extension of namespace foo.
```

This distinction between the original definition of a namespace and its extension also exists in our grammar, with two different production rules, shown in figure 3.5.

![Production rules for namespace definitions](image)

Obviously, nothing in our context-free grammar enables us to express the right constraints: parsing ambiguities systematically arise from namespace definitions. As a consequence, a post-processing filter, `afcpp-namespace`, is applied to parse forests to disambiguate these definitions.

When stripped of its implementation details, the process is actually quite simple. A top-down traversal of the parse forest is performed. When an ambiguous namespace definition is found, its name is searched for in a symbol table. If the lookup is successful, the extension branch is kept. If the lookup fails, the original definition branch is kept, and the environment is updated. A more formal definition of this is given in algorithm 3.1.

```
Algorithm 3.1 afcpp-namespace

\[
E \leftarrow \{
\]

\textbf{top-down traversal}

\textbf{for each} ambiguous definition \(d\) of namespace \(n\) \n
\textbf{do}

\begin{align*}
& \text{if} \ n \in E \text{ then} \\
& \quad d \text{ is an extension of } n \\
& \text{else} \\
& \quad d \text{ is the original definition of } n \\
& \quad E \leftarrow E \cup \{n\}
\end{align*}

\textbf{end if}

\textbf{end for}

\textbf{end top-down traversal}
```

While this filter is rather complex, compared to other components of our processing chain, it is the first one to be actually applied. The motivations for this are clear:

- Namespace definitions are traditionally very large structures in C++ programs. Since they are initially duplicated, they significantly increase the size of parse forests. Applying this
filter at the beginning of the chain quickly reduces them to more reasonable sizes, and avoids the incoming filters to process duplicate data.

- Namespace definitions do not depend on other structures of the language, which makes the early application of this filter possible. On the other hand, many filters need clean definitions of namespaces, and as soon as a filter plays with symbol names and lookups, it relies on the work done by `afcpp-namespace`: managing symbols while the parse forest is invaded with duplicate definitions does not seem very sane.

### 3.2.3 Post-processing declarations

When describing the various post-processing filters we have developed, we did mention several components dedicated to declarations. There are several reasons explaining this particular interest for declarations:

- At first, as explained in chapter 2, declarations in themselves are easily prone to produce parsing ambiguities; moreover, due to the great diversity of declarations in C++, these ambiguities are rather difficult to take care of.

- In addition to this, remember that the last part of our disambiguation chain is based on a semantic analysis step, that solves ambiguities in a general manner, by trying to determine the kind of every symbol in the program. To be able to perform this analysis, declarations must be processed enough to leave no ambiguity on the nature of the symbols being declared (aggregates, types, values, and so on...).

Therefore, to handle declarations properly, we apply several filters in sequence. Each filter of this collection manages a specific aspect of declarations, or implements a local heuristic.

Also, as it will become clear later on, splitting the post-processing of declarations into these many small pieces is not done only for simplicity or modularity purposes: some of the filters we will describe are dependent on tasks performed during previous passes. While we remain relatively free to modify the ordering of our processing chain, some of its components do not commute.

#### Declarations without declarators

Recall the piece of grammar defining the syntax of declarations, given in figure 2.1. Actually, this rule states that a C++ declaration is only the concatenation of two lists:

- A list of specifiers, which qualify the nature of the symbol(s) being declared. Some elements are always known to be specifiers (some keywords, such as `const`, `typedef`, base types, or some syntactic structures, for example class definitions), some are not.

- A list of declarators, which name the object(s) being declared, and possibly assign a value to them. As in the case of specifiers, some constructions are clearly identified as being declarators, some are not, for example a single identifier.

Since at parsing time, some chunks of the input text cannot be classified as being specifiers or declarators, a declaration can often be read in several distinct ways, with respect to the grouping of tokens into the lists mentioned above. Program 3.4 gives some examples of ambiguous and non-ambiguous groupings.

Declarations are most of the time parsed with additional ambiguities, but the purpose of `afcpp-declaration` is only to handle this grouping issue.

To this end, we only consider ambiguous declarations, and apply a very simple rule: their declarator list should not be empty; branches of declarations that do not satisfy this constraint are removed.
Program 3.4 Grouping in declarations

```c
int foo = 0; // Not ambiguous.
int foo, bar; // Not ambiguous.
foo bar; // Ambiguous: ([foo bar], []) or ([foo], [bar]).
typedef foo bar; // Ambiguous: ([typedef foo bar], []) or ([typedef foo], [bar]).
foo bar = 0; // Not ambiguous.
foo bar, baz; // Not ambiguous.
class A {}; // Not ambiguous.
class A [] a; // Ambiguous: ([class A [ ] a], []) or ([class A [ ]], [a]).
```

Of course, this is not the case for all declarations (typically, a class declaration does not have any declarator), but the declarations for which this rule does not apply do not suffer from any ambiguous grouping, and are therefore not seen during this stage.

The processing performed by `afcpp-declaration` is also described in algorithm 3.2.

Algorithm 3.2 `afcpp-declaration`

```
top-down traversal
  for each ambiguous declaration d do
    for each branch b(\(b_s\), \(b_d\)) of d do
      \(b_s\) is the list of specifiers carried by \(b\)
      \(b_d\) is the list of declarators carried by \(b\)
      if \(b_d = []\) then
        remove \(b\) from \(d\)
      end if
    end for
  end for
end top-down traversal
```

Ambiguous declarators

The previous filter did handle the grouping of elements in declarations. Unfortunately, declarations suffer from other kinds of ambiguities.

Program 3.5 Another simple declaration

```c
typedef int foo; // Parse forest extracted from the foo part of the declaration.
```

Among other things, the identifier part of each declarator always produces several parses, like any other identifier: the grammar distinguishes several types of identifiers (expressions, types, classes, enumerations, and so on...), but there are no clues at parsing-time of how this classification should be performed. A partial parse forest showing an ambiguous declarator is given in figure 3.6; it has been generated from program 3.5.

While we have chosen to handle this identifier issue with a general approach, by performing a whole program analysis to determine the kind of every symbol, this approach does not hold for declarators:

- Declarators introduce new symbols in the program. Of course, the associated identifier is
not of a particular nature: in practice, a declarator identifier is simply marked as being an expression identifier.

- In order to perform our global program analysis, we need to be able to establish a reasonable amount of knowledge by using declarations. To this end, although we can work with partially ambiguous declarations, we cannot afford fuzzy declarators.

Anyway, once that `afcpp-declaration` has ensured a correct grouping of symbols in all declarations, fixing the declarators is quite easy. A filter dedicated to this task, `afcpp-declarator`, performs the algorithm 3.3.

### Algorithm 3.3 `afcpp-declarator`

```plaintext
top-down traversal
  for each declaration \(d(d_s, d_d)\) do
    \(d_s\) is the list of specifiers carried by \(d\)
    \(d_d\) is the list of declarators carried by \(d\)
    for each declarator \(x\) of \(d_d\) do
      keep the expression branch of \(x\)
    end for
  end for
end top-down traversal
```

### Ambiguous sequences of specifiers

When both `afcpp-declaration` and `afcpp-declarator` have been applied to the parse forest, declarations may still contain some ambiguities that can be processed at a local level.

A very curious problem arises in lists of specifiers, when qualified names are being used, and, once again, it is a matter of grouping.

Actually, many of the problems we encounter in declarations have the same root: the elements in a list of specifiers are not separated by any kind of token. This may sound quite harmless, but in practice, many forms of lists that involve qualified identifiers can be read in several ways. Some examples are given in program 3.6.
Program 3.6 Qualified names in lists of specifiers

```c
typedef foo bar;  // Not ambiguous.
typedef foo::bar baz;  // Ambiguous: [foo::bar] or [foo, ::bar]?
typedef foo::bar::baz qux;  // Ambiguous: [foo::bar::baz] or [foo, ::bar::baz] or ...
```

As shown by this sample program, the size of the parse forest can quickly grow up as soon as some complex identifiers are being used. However, filtering such lists can be done with a very simple criterion: it seems pretty clear that, while a list of specifiers may be composed of many elements, primarily keywords (`static`, `const`, `virtual`...), it should not contain more than one type specifier. From there, we derive a new filter called `afcpp-specifier`, that applies the algorithm 3.4.

Algorithm 3.4 `afcpp-specifier`

```plaintext
top-down traversal
  for each ambiguous list of specifiers \( l \) do
    for each ambiguous branch \( l_i \) of \( l \) do
      compute the number of non-trivial specifiers \( k_i \)
    end for
    keep \( l_i \) such as \( k_i = 1 \)
  end for
end top-down traversal
```

By applying this constraint, we are able to both disambiguate the lists of specifiers seen above, and reject some invalid inputs as well.

3.3 Semantic analysis

When a parse forest has been processed by all the components seen above, it can be passed to the last filter, `afcpp-disambiguate`, that finishes the disambiguation stage by performing a global analysis of the input program.

This filter is rather complex when considered with all its details, but it is derived from a very basic and natural idea. At this stage of processing, all the remaining ambiguities are related to identifiers; the question being to know whether a given symbol is a type, or class, or value, etc. Therefore, to complete the reduction of the parse forest, the following two-stage strategy can be applied:

1. Traverse the parse forest, and gather informations from declarations. A collection of environments is built; each environment encodes a namespace of the input program, and associates in this namespace symbols to their kinds.

2. Traverse the parse forest, and for each ambiguous node, check its branches, and keep the correct candidate. In particular, on each symbol, perform a lookup in the relevant environments to check if its current interpretation is correct.

More details on this filter are given further in this section. After the second pass, the filtering has either failed\(^2\), or the parse forest has been completely reduced. Pay attention, though, to the fact that we dot not yet guarantee any kind of correctness on a program that has passed the filtering process.

\(^2\)This means that the input program was ill-formed.
3.3.1 Classifying declarations

Above, we described the first pass of afcpp-disambiguate as a traversal that constructs the environments of a program out of the declarations. This description is not quite correct; in practice, all declarations are not systematically processed during this first pass.

Actually, this filter is implemented in two stages to enable us to manage properly constructions such as classes, where symbols may be used far before their definition (typically, public methods make use of private attributes defined later, see program 3.7).

Program 3.7 A class declaration

```c
class Complex
{
  public:
    Complex (float re, 
              float im) : re_ (re), 
                  im_ (im) 
    {
    }

    inline float getRe () { return re_; }
    inline float getIm () { return im_; }

  private:
    float re_; 
    float im_; 
};
```

On the other hand, to be able to perform a correct analysis of the input program, there are many declarations that should not contribute to build environments during the first pass. This is the case for local declarations, since a local symbol cannot be used before its declaration.

Finally, we have two constraints on declarations that make it possible to determine which declarations should be considered during the first stage of this filter:

- In some specific constructs, symbols may appear before their declaration. This is the case for namespaces and classes, but not for local declarations.
- We need to build environments during the first processing pass, and reuse them during the second stage. Therefore, we have to find a way to name uniquely the symbols inserted into these environments. In C++, without any additional work ($\alpha$-conversion, etc.), some symbols can be uniquely named, some cannot.

Luckily enough, these two sets nearly match: we need a preprocessing for constructions such as classes and namespaces, and these structures introduce named scopes that make the construction of our environments possible.

Some constructions that need a preprocessing cannot be associated easily with a unique qualified name, mainly local classes and anonymous classes. These have not yet been addressed in our framework.

3.3.2 Reading partially ambiguous declarations

The previous classification provides the very basic guideline that stands behind the last filter. We now know which declarations should be used to construct our “static” environments, and which
declarations should only enrich these preexisting environments during the final traversal of the parse forest.

Of course, the filter is built on top of the previous disambiguation stages. At this point, while we still have a very ambiguous parse forest, we have reasonably reduced declarations; they are still ambiguous, but usable. In practice, these declarations give only partial information, but are verbose enough to determine the kind of every symbol being declared (whether it is a type, class, value, etc.).

Algorithm 3.5 \texttt{afcpp-disambiguate}

\begin{algorithm}
\begin{algorithmic}
\STATE \textbf{[build environments]}
\STATE \textbf{top-down traversal}
\FOR {each definition of namespace or class \textit{n}}
\STATE $E_n \leftarrow \emptyset$
\ENDFOR
\FOR {each declaration \textit{d} of symbol \textit{s}}
\STATE \textit{s} is declared in namespace \textit{n}
\STATE \textit{s} is of kind \textit{k}
\STATE $E_n \leftarrow E_n \cup \{s: k\}$
\ENDFOR
\END top-down traversal

\STATE \textbf{[disambiguate]}
\STATE \textbf{top-down traversal}
\FOR {each ambiguous node \textit{a}(\textit{a}_1,a_2,\ldots,a_n)}
\STATE keep \textit{a}_i such as \texttt{disambiguate}(\textit{a}_i) is successful
\ENDFOR
\FOR {each symbol \textit{s} seen with kind \textit{k}}
\STATE find namespace \textit{n} in which \textit{s} is declared
\IF {$E_n \vdash s: \textit{k}' \neq \textit{k}$}
\STATE fail
\ENDIF
\ENDFOR
\FOR {each local declaration \textit{d} of symbol \textit{s}}
\STATE \textit{s} is declared in namespace \textit{n}
\STATE \textit{s} is of kind \textit{k}
\STATE $E_n \leftarrow E_n \cup \{s: k\}$
\ENDFOR
\FOR {each scope in namespace \textit{n}}
\STATE save and restore $E_n$ properly
\ENDFOR
\END top-down traversal
\end{algorithmic}
\end{algorithm}

Algorithm 3.5 is applied by \texttt{afcpp-disambiguate}. Notice how the second pass of the filter is a recursive checking process\footnote{Actually, this filter associated with the previous passes is a refined version of the naive disambiguation algorithm. Disambiguation could be performed simply by checking every possible tree with an algorithm similar to algorithm 3.5, but the number of possible trees is exponential in the number of ambiguities.}; for each ambiguity, the different possible branches are disambiguated, but this is expected to fail on all branches but one. Failures are raised by incorrect subtrees, mainly identifiers seen with a wrong kind.
Chapter 4

Conclusion and further work

In this report, we have presented the very early stages of development of a framework for program transformation in the C++ language.

We have described the additional software components that have been developed to work altogether with the meta-tools Sglr and Stratego, as well as our original approach to the syntactic analysis of C++, a non-deterministic parser assisted by a bundle of disambiguation filters.

Yet, while the results are promising, there are still many limitations to our system, and much work to be done to achieve our primary goal of automatic derivation of active libraries.

4.1 Limitations

Most limitations of our system are related to the filtering process, where some constructions of the language are not yet properly handled. Among constructions of this kind, most problematic are:

- Anonymous classes, which do not fit yet in our simple name lookup model.
- Class declarations local to functions, for the same reason.
- Template-based constructions, probably the most unsettled point, for a general solution would require to implement the template instantiation mechanism of C++, a large task indeed.

4.2 Further work

Apart from the filtering stage, there are many more general issues that have not yet been addressed:

- The most critical point is the C pre-processor. Until now, we have not yet taken into account this stage, but simply applying our transformations after the pre-processing, as the compiler does, is not a satisfactory solution.

  When transforming programs, in particular when these are intended to remain human-readable, we cannot afford to let the pre-processor pollute the resulting source code, by, for example, copying into each file the definitions from the C++ standard library.

- Still in the context of transformations producing human-readable programs, we need a processing chain able of preserving comments in the code. This is actually not the case, since the implosion of parse trees into abstract syntax trees strips all layout information. Working solely on abstract syntax trees is, obviously, not the best suited method.
• Last, as we explained it in the previous chapter, our syntactic analyzer (parsing plus dis-
ambiguation) does not guarantee the syntactic correctness of the input programs.
While this deficiency was acceptable at the beginning of the project, with the C++ compiler
acting ultimately as an oracle, this is, of course, to be corrected sooner or later.
Bibliography


