Centaur: A generic framework simplifying C++ transformation

Cedric Raud

Technical Report n^o0823, July 2008
revision 1848

English: The C++ standard grammar was not thought to be easily parsable so its use in the context of program manipulation can be compared to the complexity of ASTs it generates. The aim of Centaur inside the Tranformers Project is to propose a generic framework to manipulate and digest this AST: program transformations are simplified thanks to an easier access to the parse tree data and its annotations. With this library, repetitive and error-prone tasks like enumerating a container's elements or the parent classes lookup of a class will be factored by a function set corresponding to an extensible and modular model.

French: La grammaire du standard du C++ n'ayant pas été conçue pour etre aisément analysable, son utilisation dans le cadre de la manipulation de programme est comparable à la complexité des ASTs générés par celle-ci. Le rôle de Centaur au sein du projet Transformers est ainsi de fournir une infrastructure générique permettant de manipuler et de synthétiser cet AST : les transformations de programmes sont simplifiées grâce a un accès plus aisée aux informations contenues dans l'arbre syntaxique et ses annotations. Grâce à cette bibliothèque, les tâches répétitives et souvent génératrices d'erreurs, comme l'énumération des éléments d'un conteneur ou la recherche des classes parentes d'une classe, seront factorisées par un ensemble de fonctions correspondant à un modèle modulaire et extensible.

Keywords
Transformers, C++, Centaur, Stratego, AST, Program transformation

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
spycam@lrde.epita.fr – 200807-Seminar-Raud

Copying this document

Contents

Chapter 1

Introduction

Belonging to the domain of software engineering, program transformation is the operation of translating a program into another program. Even if the technology used by program transformation is globally the same that the one that can be found in compilers, the result of these techniques can be far more various and can answer to a lot of different needs.

For example, we can use program transformation to derive a program from a specification or to track and enhance the performance of a piece of code. We can also simplify the rewriting of large piece of code by using automatic program refactoring. We can explode a program in small pieces in order to analyse the components of the system like in reverse engineering. But one of the main application of program transformation is certainly the extension of a language that permit to adds features or simplify the writing of high-level constructs.

This document is presented as follows: the next chapter is a panorama of the principal existing solutions in the domain of source-to-source program transformation. Then, there will be a global overview of the Transformers framework and its architecture. Finally the Centaur framework will be introduced and detailed.

Acknowledgments

- Nicolas Pierron for his help on Stratego/XT,
- Roland Levillain and Warren Seine for the review of this technical report.

Chapter 2

State of the art of program transformation

2.1 Codeboost

2.1.1 Presentation

Website: http://www.codeboost.org

Developed as a part of the Sophus project which experiments the use high-level abstractions for numerical applications, Codeboost (Bagge, 2003) is a tool for source-to-source transformation and optimisation of C++ programs.

Codeboost is specially designed to answer to the Sophus needs and to be a bridge between the Sophus style of programming and current compiler technology. To compensate for the lack of optimisations of the compilers, they have written their owns system in Stratego to allow domain-specific optimisations. For example, users can write rules directly inside C++ programs so that permit Codeboost can apply the corresponding transformations on the abstract syntax tree.

The following piece of code, added to a C++ program, will be automatically understood by Codeboost.

```
void rules()
{
  int x;
  int y;
  int z;

  assoc: (x + y) + z = x + (y + z);
}
```

Figure 2.1: Simple user defined rule

The user will next be able to use this rule into his code by adding “assoc:” at the beginning of the line. The rewriting of the AST is shown in Figure 2.2.

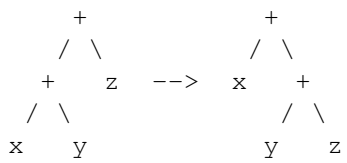

Figure 2.2: The transformation of the AST

Obviously, it is also possible to write the transformations manually directly in Stratego.

2.1.2 Differences with Transformers

- The main difference between Codeboost and Transformers is that Codeboost uses openC++ to parse sources. This implies that they do not support a fully compliant C++ grammar (ISO/IEC, 2003) : namespaces, pointers to functions, and inheritance are not handled to keep the implementation concise and manageable.
- User defined rules is a feature that is not planned to be implemented into Transformers.
- The documentation is not available.

2.2 Dryad

2.2.1 Presentation

Website: http://www.program-transformation.org/Stratego/TheDryad

Also written in Stratego, Dryad (Bravenboer (2005)) is a collection of tools for developing transformation systems for Java programs. The original capability of this project is that it can handle Java sources as well as Java bytecode.

The project is built as a library which embed a large panel of functions useful for developing systems in Stratego and have a type checker built in.

2.2.2 The model

They have created their own model for representing Java source and bytecode in Stratego and this model comes with a well documented API.

They have succeeded in having an object oriented model for Java with a linked structure. Thus, we can find in the model an abstract class that represents a Java class which is implemented by bytecode-class and source-class. Thanks to the links between the objects of the model, we have methods which simplify the traversal of the structure: get-superclass, get-methods, get-fields, get-formal-type-parameter, etc...

Nonetheless, they don’t use references between the different objects but a big hash table called the “repository”. As Martin Bravenboer, one of the main developer of Dryad, says « Dryad uses one big hashtable for all fields of objects, which is a nightmare for garbage collection ».

2.2.3 Differences with Transformers

- The type checker is working.
- The transformation part in complete
- The model is not conceived to be easily extensible or reusable: this project doesn't target language extension.
- The project parses Java only.

2.3 JastAdd

2.3.1 Presentation

Website: http://jastadd.org

JastAdd (G. Hedin, 2003) is an open source Java-based system for compiler construction. In other words, these tools allow to write a compiler or related tools like analyzers or transformations tools in a language based on Java.

JastAdd is centered around an object-oriented representation of the abstract syntax tree and uses reference variables to link together the different parts of the tree.

2.3.2 Differences with Transformers

- JastAdd doesn't provide tools to handle C++.
- They use references to link the elements of the tree.
- The initial purpose is not to achieve program transformation.

2.4 Pivot

2.4.1 Presentation

Website: http://parasol.tamu.edu/pivot

Developed by the creator of C++ himself, Bjarne Stroustrup, the Pivot (B. Stroustrup, 2004) is a framework for static analysis and transformation of C++ programs. The project presents a set of tools for converting between IPR and XPR but also general traversal and transformation tools.

2.4.2 IPR

Acronym for "Internal Program Representation", the IPR is the core of the Pivot project. With the exception of macros, IPR is a fully general typed abstract syntax tree representation of the whole language. Hopefully, the model will stay valid with the future C++ standard because this model is already prepared for C++0x facilities, notably concepts. Contrary to the previous projects the model is here implemented in C++ so there is a large use of the object oriented paradigm in the representation of the C++ trees. The model has also an unified type system and is potentially standard according to Bjarne Stroustrup. The interesting feature is that the model can represent incomplete or erroneous programs.

2.4.3 XPR

Next to IPR, XPR, "External Program Representation", is a compact, user-readable, portable representation of IPR. Designed to reflect the inner syntax of the C++, XPR is a language which can be used in systems using IPR to model C++. Its goal are to be simple and fast to process while being human-readable. One of the main constraint was also to permit parser to analyze it quickly.

2.4.4 Differences with Transformers

- The project is not publicly available.
- The source code is not available.
- The documentation is incomplete (we don't know if the project is really usable, if they have their own preprocessor, etc...)

Chapter 3

The Transformers framework

3.1 The idea

The idea of the Transformers Project comes from the expertness of the LRDE in the writing of statically-typed libraries which use a lot a meta-programming techniques. Indeed, if meta-programming in C++ offers us a new way to write solid, generic and maintainable code, its readability is inversely proportional to the amount of user-defined types used within the program.

The situation looks paradoxical: we want generic and maintainable program and we have source codes that become harder and harder to read, write and understand. Furthermore, meta-programming makes the compilation process become really heavy because of the computations made at compile-time.

3.2 A C++ source to source transformation framework

The work on the C++ transformation framework was initiated at LRDE by Robert Anisko in 1999. Conceived as an environment for C++ source to source transformation, the main goal of Transformers is to simplify the writing of such code and to help programmers in their C++ development.

As it will be detailed in the next section, the source code is processed, parsed, transformed, pretty-printed and then unprocessed. The resulting code can then be read and modified again by programmers and thus must be human-readable. It should also respect the original coding style.

The final goal of Transformers is to help the writing of language extensions and provide usable tools for C++ program refactoring.

3.3 Architecture

In order to have a better vision of the Transformers mechanism and where Centaur takes a role in the framework, here is a general overview of the Transformers pipeline. All the different steps detailed below are connected together with unix pipes by the Transformers front-end, `parse-cxx`.

3.3.1 C++ Source

The input source can be classical C or C++ source file or header file. It may include `#INCLUDE`, `#DEFINE`, and other preprocessors directives. The only limitation is that the file must respect the C++ standard grammar.

3.3.2 revCPP

First, the source code is applied to revCPP (Sigoure and Hocquet, 2008), a reversible C++ preprocessor released under the GNU GPL and developped inside the project. The particularity of this implementation of CPP is that it adds extra-comments to be able to reverse the preprocessing.

3.3.3 Parsing

Then, the essential step of the pipeline is the parsing of the code. The tool used in this section is a generic SGLR parser which implements scanner-less generalized LR parsing. The grammar is described using SDF, a modular syntax definition formalism. Our C++ grammar is downloadable on the Transformers website [1]. Even if we use a generalized parser, we have the possibility to keep the ambiguities produced by the parser and to have a parse tree instead of a single tree. That permits to use a separate tool to do the disambiguation job.

3.3.4 Disambiguation

The disambiguation proccess aims at killing the invalid branches of the parse tree with the help of the attribute grammar engine, `sdf-attribute`. Attribute grammars are parse-tree annotations permitting to link semantic information with syntax definition. They are not currently supported by Stratego but Transformers embedded its own attribute grammar engine (Pierron, 2007).

3.3.5 Program transformations

Implemented in Stratego, the program transformation step consists in manipulating the abstract syntax tree in order to translate the original program into a new one. The Centaur framework will play an important role here because today the tree is too big to be easily transformed. Writing transformations with Centaur will consist in importing the Centaur library into a Stratego piece of code and using the functions provided to manipulate and analyze the C++ tree.

3.3.6 Pretty Printer

The pretty-print step takes the tree and reads the information contained in its nodes to display C++ code back.

3.3.7 revCPP

revCPP plays the same role as mentioned before but backward: the file in unprocessed according to the extra-comments.

[1] http://transformers.lrde.epita.fr

3.3.8 Improved C++ source

We now have a valid C++ source code with the original macros and coding style of the developer.

Chapter 4

Centaur

4.1 A Stratego Library

As presented in the previous chapter, Centaur is a large library of functions useful for developing C++ transformation systems in Stratego. If this library is crucial to achieve complex transformations, its target is not to design the transformations themselves. Centaur will only provide a model of the C++ and all the functions needed to manipulate the abstract syntax tree. That is why building Centaur correctly is an important task: we have to find a good balance between the complexity of the C++ tree and a easy-to-handle design.

4.2 The C++ grammar

Since the idea is to offer the possibility to transform his own code, the principal constraint that we set is to remain as close as possible to the grammar given in the C++ standard. We can reasonably think that the code produced in an average project stays within the borders of this grammar.

The main disadvantage of this choice is that the C++ grammar was made for LALR(1) parser and is not easy to manipulate in the context of transformation writing of static analysis. Indeed, nodes of the C++ standard grammar encapsulate a lot of information and there is different nodes for similar constructs.

For example, a classical "return 42;" produces the corresponding tree of the Figure 4.1.

```
Statement-p(
  [ JumpStatement(
      return(
        Some(
          Expression2(
            [ ConditionalExpression(
                LogicalOrExpression(
                  LogicalAndExpression(
                    InclusiveOrExpression(
                      ExclusiveOrExpression(
                        AndExpression(
                          EqualityExpression(
                            RelationalExpression(
                              ShiftExpression(
                                AdditiveExpression(
                                  MultiplicativeExpression(
                                    PmExpression(
                                      CastExpression(
                                        UnaryExpression(
                                          PostfixExpression(
                                            PrimaryExpression(
                                              Literal(
                                                IntegerLiteral(
                                                  INTEGER-LITERAL("42")
                                                )
                                              )
                                            )
                                          )))))))))))))))))
            ]
          )
        )
      )
  ])
```

Figure 4.1: Simple user defined rule

4.3 Implementation

To handle such large tree, one of the solutions would be to use concrete syntax but there are too many ambiguities to rely on it.

The closest existing solution that fits our needs is the model used into Pivot, the IPR. Nevertheless, its large use of inheritance makes it difficult to translate into Stratego. Mixing the Dryad use of OOP in Stratego and the simplicity of the IPR of Pivot would be the optimal solution to our needs.

Chapter 5

Conclusion

Even if the objectives of Centaur are very (maybe too) ambitious for this seminar, integrating this library into Transformers would be an important jump in the simplification of program transformations into this project.

Chapter 6

Bibliography

Anisko, R., David, V., and Vasseur, C. (2003). Transformers: a C++ program transformation framework. Technical Report 0310, LRDE.

B. Stroustrup, G. D. R. (2004). The pivot framework: Design and implementation (http://parasol.tamu.edu/pivot/presentations/DosReis.pdf).

Bagge, O. S. (2003). Codeboost: A framework for transforming c++ programs (http://www.codeboost.org/papers/exam-otto-03.ps).

Bravenboer, M. (2005). Infrastructure for java transformation systems (ftp://ftp.stratego-language.org/pub/stratego/SUD05/dryad.pdf).

G. Hedin, E. M. (2003). The jastadd system - an aspect-oriented compiler construction system (http://www.cs.lth.se/%7Egorel/publications/2003-JastAdd-SCP-Preprint.pdf).

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

Pierron, N. (2007). Formal definition of the disambiguation with attribute grammars. Technical report, EPITA Research and Development Laboratory (LRDE).

Sigoure, B. and Hocquet, Q. (2008). revCPP a reversible C++ preprocessor. Technical report, EPITA Research and Development Laboratory (LRDE).