Proceedings of the 7th International Conference on Applied Informatics
Eger, Hungary, January 28–31, 2007. Vol. 1. pp. 147–154.

Extension of GCC with a fully manageable reverse engineering front end

Csaba Nagy

Department of Software Engineering, University of Szeged
e-mail: Nagy.Csaba.5@stud.u-szeged.hu

Abstract

In the open source community one of the most popular compilers is GNU GCC. It is a very complex and robust compiler, but because of its working mechanism it has no ability for special transformations like interprocedural optimizations.

A typical compiler has a three sided construction. It has a front end for analyzes and for building an abstract internal representation of the program, a middle for transformations (e.g. optimizations), and a back end for final code generation. However there are smaller but very useful projects for only front/middle/back ends too, it seems possible to achieve a more effective compiler by extending GCC with a front end which is capable of running special algorithms.

This paper shows one solution for this extension. The described method is based on using Columbus/CAN instead of GCC's front end, and because Columbus has a well-structured schema for representing C++ sources, by this extension the compiler will have the ability to execute those special transformations on the code before the compiling phases. Furthermore this technique opens the possibility to connect other front ends – like Edison Design Group's C++ front end – to GCC and achieve a more powerful compiler, for example in code size optimizations.

This approach has been tested on small C projects like bzip2 as a real-world system, and on parts of the official Code-Size Benchmark Environment (CSiBE) of GCC.

1. Introduction

For big commercial or even non-commercial projects it is very important to produce a fully optimized binary code for final release. While developers are working on the implementation they are usually not able to observe every single optimization possibilities so final source analyzes are necessary. The main goal for these analyzes, can be the better performance for running speed, binary size or anything

relevant for the target system like the energy consumption for a laptop or a handheld device (e.g. PDA). For companies or developers who would like to produce a project as effective or perfect as possible, these aims are particularly relevant.

The *GNU Compiler Collection* GCC [3] is a set of programming language compiler which already contains many optimization algorithms but in several cases its structure incapacitate it for doing special kind of optimizations. One typical example is the *Interprocedural Analysis* (IPA) which is based on the main idea of producing algorithms which work on the entire program, across procedure or even file boundaries. For a long time the open source GCC had no powerful interprocedural methods because it is structure was optimized for compiling functions as units. Nowadays new IPA framework and passes were introduced by the *IPA branch* but these are still under heavy development [8] and GCC will still have the weakness that it is taking one source input as a compilation unit.

This paper introduces a novel possible solution for modifying the structure of the compiler, to make it capable for doing new optimization methods like interprocudral optimizations.

There are useful commercial and freeware applications for doing different analyzes and optimizations on the source code but usually these applications can not act as a compiler. One of these applications is *Columbus/CAN* (released by FrontEndArt Ltd.) [7, 1] which is a reverse engineering framework application providing environment for parsing, analyzing, filtering and exporting information extracted from source files. Because Columbus has a well structured *schema* [6] for representing the whole C/C++ source code of a project, it is also good for doing kind of transformations on the code that would be impossible in GCC optimization phases. Just to mention one example the described IPA optimizations could be realized on Columbus representation even over file boundaries.

As Columbus/CAN front end is widely extensible through a well documented *Application Programming Interface* (API) it seems possible to link GCC and the FrontEndArt Ltd's reverse engineering framework together, and have an "extended compiler" with a fully manageable front end. This linkage presented in this paper, is a first try for extending GCC with an other front end application, so it opens the possibility to link other source analysis softwares and front ends to the compiler as well. In this way we can achieve a more powerful compiler, for example in optimizations.

2. Overview

2.1. Construction of GCC

A compiler – like GCC – has a typical three sided construction [2]. It has a *front end* for parsing and analyzing the source code and for constructing an *abstract syntax tree* (AST, also called *abstract syntax graph*, ASG) as an *intermediate representation* (IR, also called *intermediate language* or *intermediate representation language*, IL, IRL) of the code. It has a *middle end* for doing transformations

(e.g. optimizations) on the internal representation builded before and for preparing this representation for final code generation which is realized by the *back end*. (Figure 1.)

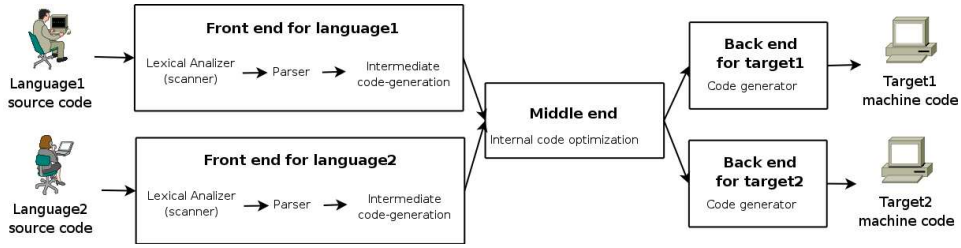

Figure 1: Construction of a compiler

The C++ front end of GCC first does preprocessing (handling macros, etc.) on the input file, breaks it into tokens and parses it. During this parsing it does lexical and syntax analysis while constructing the AST as well. This abstract tree is the highest level IL of GCC, containing language specific elements of C/C++. Its internal representation is based on the `tree` structure which is used for describing lower level ILs too. So the front end can easily transform this AST to GENERIC and lower it to GIMPLE and *Tree-SSA* form which representations are language independent intermediate languages to store different kind of information for various optimization passes realized by the *middle end*.

The *middle end* executes language independent transformations and optimization algorithms on different ILs. First on the GIMPLE and finally on the *Register Transfer Language* (RTL) representation levels which is the lowest level, very close to the final assembly that is generated by the *back end*. (Figure 2.)

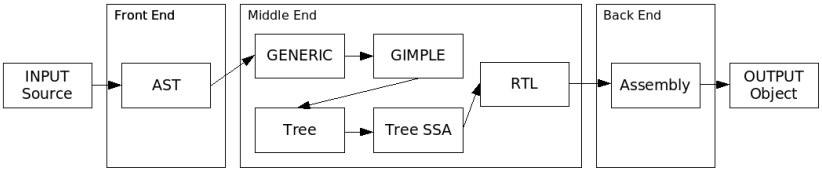

Figure 2: The internal representation languages used by GCC

2.2. About Columbus framework

Columbus reverse engineering framework has been developed by FrontEndArt Ltd. in a cooperation between the Research Group on Artificial Intelligence in Szeged and the Software Technology Laboratory of the Nokia Research Center. It

is able to analyze large C/C++ projects and calculate different metrics on them [9].

The framework contains many applications for different purposes so in this section the described applications are only the emphasized ones for the extension:

CANPP *C/C++ ANalyzer-PreProcessor* is a command line tool for preprocessing given source files. It is input is a `.c` or `.cpp` C/C++ source file and output is a preprocessed `.i` file. Using parameters many configuration arguments can be passed like location for include files, macrodefinitions, filters, etc.

CAN The preprocessed `.i` file can be passed to *C++ ANalyzer* which is a command line program for source code analyzation. This tool constructs for source code representation the *abstract syntax graph* (ASG) using the rules defined by the *Columbus Schema* [6]. The output of this application is a `.csi` file which is a *schema instance*, a binary output of the ASG representing the input source.

3. The extension

For doing the extension first an entry point must be located in GCC where we can join in the compilation passes and force the compiler to use source information retrieved from Columbus. On the Columbus side it seems to be evident to use the *schema instance* (`.csi`) which contains all important data about the source file in an easily readable form using the Columbus API. Because this Columbus ASG is best comparable to the GCC C++ front end's AST, the best solution seems to be a transformation between the two representation graph. GCC builds the AST during parsing and later it does transformations on it. These transformations are first language specific modifications (like the creation of default constructors and template instances), and later transformations by the *genericizer* which transforms this tree to the language independent GENERIC form. Because the modifications realized by the front end on the AST level might be necessary on the Columbus ASG too, the entry point must be before these modifications.

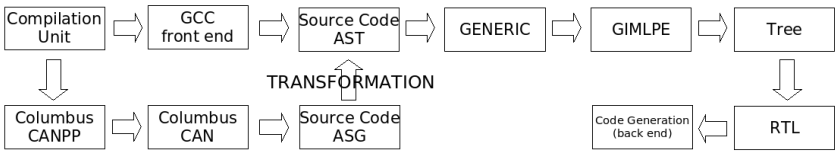

Figure 3: The entrypoint to connect Columbus and GCC compilation passes

One way for this extension is to substitute the parser with a *Columbus-GCC converter* which transforms the *Columbus Schema* to the GCC C++ AST. After

doing this conversion GCC can take back the control and continue with compilation passes to finish the compilation using the AST constructed from a Columbus *schema instance* (Figure 3).

3.1. Theoretical background

The theoretical background of the Columbus-GCC ASG conversion can be described from a mathematical and a programing view either.

For the a mathematical view, first we have to give a definition for identical Columbus ASG and GCC AST graphs. So suppose that a $C(V_C^p, E_C^p)$ graph represents a Columbus ASG and a $G(V_G^p, E_G^p)$ represents a GCC graph, where p is the input program. In this case we can easily give a "lazy" definition when we define the two graph identical if they produce the same output for same input information. However this definition works well for checking that two graphs are identical or not, it does not contain any information for doing the transformation. So we have to define a "strict" definition as well. This definition first defines small identical graphs for semantic elements of the source language, and later it defines recursive rules for equality.

Using the definitions, converting a Columbus ASG to GCC AST is a $\phi_i : C \to G$ partial mapping between identical representation graphs, where C domain is the set of *schema instances* and G codomain is the set of possible instances in *abstract syntax tree* form of GCC. As a $g_i \in G$ *AST instance* can be retrieved from many $c \in C$ *schema instances* this mapping is *non injective* and because several cases $g_j \in G$ *AST instance* has no parent in C it is *non surjective* as well. For addition the number of ϕ_i mappings is about n because the conversion can be realized in more possible way so for one $c \in C$ instance more $g \in G$ structure may exist.

From the programing view, this conversion means a transformation between two really different data structures which were designed for storing the same information. The base of the transformation is a given *schema instance* (`.csi`) file which stores a Columbus ASG on a high level object orientated IL. The target of the conversion is the AST IL of GCC which is a lower level language, based on the `tree` internal structure of the compiler. For doing this conversion we should walk over each existing node of *Columbus Schema* and find the node with same meaning in the GCC AST. If this node exists, we have to create it, fill the fields of it with corresponding data, and connect the edges to other already builded nodes.

As *Columbus Schema* is quite well documented, nodekinds and connections between them can be easily recovered in spite of GCC, which has a very weak documentation and usually the source is the only help for gathering information about the AST. Columbus represents every statement with a class usually with the same name of the represented nodekind. Connections between these nodes (or classes) are described by aggregations and associations.

GCC on AST level represents the source using a structure named `tree`. This structure is based on a record type definition with various fields for storing different facts about the statements (e.g. each node has a TREE_CODE for determining which kind of statement it is).

3.2. Implementation and technical details

For development a GCC source snapshot was taken from the mainline version 4.2.0. Columbus CAN and CANPP were version 3.6 beta with the corresponding libraries for Columbus API.

Because doing the conversion from Columbus ASG to GCC AST, both of the Columbus API and GCC API calls are used, it resulted that actually a robust C++ project (Columbus) is linked to an other much more robust C project (GCC). For doing this linkage, the *Columbus-GCC Converter* source is sperated in a directory inside the GCC source tree with an own *Makefile*. This file contains instructions invoked when GCC is compiled with `make g++` command, to compile the C++ front end as well. The project uses the GCC include files inside `extern "C"` brackets and on the other hand for the C++ Columbus libraries it uses *g++* instead of *gcc* for linking final binaries.

For the invocation of this nested converter the following new flags were added to the compiler: *-fnoparse* and *-fcsi-file=<filaname>*. The first flag works for skipping the parser part of the compiler, and the second acts for invoking the init function of the *Columbus-GCC Converter*. This init function is called by the *front end* after the call for the parser but before finishing the AST.

The conversion algorithm itself is based on the *pre-order visitor* method offered by the Columbus API. The theory of this method is described in the *visitor design pattern* [4] which makes it possible to walk through the source code and extract information by "visiting" each node exactly once without modifying it. The systematic for visit order is a pre-order traversal that is a tree traversal algorithm defined as follows: visit the root first; and then do a preorder traversal each of the subtrees of the root one-by-one in the given order.

The init function of the *Columbus-GCC Converter* executes a *visitor* on the ASG for the input *schema instance* file, and when this visitor visits a not-yet-converted node the visit method invokes a conversion function for the given kind of source element. This conversion will result with the corresponding `tree` structure for the given Columbus node and even link this structure to the already builded GCC AST. For linking the new `tree` node to the proper position inside the builded AST, a stack stores on which level of the AST and on what kind of node works currently the converter.

Nevertheles, a conversion function builds the whole subtree for the given node of the *schema instance*. The reason for it is simple because for a complete transformation the algorithm has to fill the connection edges of the GCC node as well, and it can do it only after building the connected nodes too (e.g. for a function node next to the function declaration the parameters and the function body must be builded as well).

This may result that the conversion reaches and converts a schema node before the visitor visits it. In these cases it should be monitored that one node is converted exactly once. For this checking the converter stores pointers to already transformed nodes in an associative array which takes the Columbus nodeid as a key. Using this array the conversion can be much more faster, especially for type nodes because

the construction of complex data types is realized only once.

Thanks to the *visitor* algorithm and the conversion functions, the converter transforms all nodes of the source tree in a simple, logical way even for complex node types like *Class* and *Function*.

4. Experiments and further improvements

In the current state of the implementation the converter can handle and transform nearly all lexical elements of the full C syntax and many nodes from the C++ extension. It can deal with functions, classes, type declarations or definitions and with main sequence structures (iterations, selections) as well.

The current implementation for this "extended compiler" was able to compile the 1.0.4 version of *bzip2* as a C project which has a complex semantic structure with about 8000 lines of source code. Nevertheles, it successfully compiled small C++ source files and parts of the GCC's official Code-Size Benchmark Environment (CSiBE) v2.1.1 [5] which is an environment developed especially for code-size optimization purposes in GCC. It contains small and common used projects like zlib, bzip, parts of Linux kernel, parts of compilers, graphic libraries, etc.

Because Columbus/CAN is a robust source analyzer with many features, still under heavy development, during the implementation process I reported many bugs to the developers. Fortunately I could keep a live contact with the developers, and these bugs were eliminated soon, but in several cases – as a workaround – I had to modify the original source code to make it acceptable for the analyzer.

As a further improvement the implementation for the template package of *Columbus Schema* should be realized, but with this implementation one must wait for a new Columbus/CAN release, because the current available version (v3.6 beta) of it does not support perfectly the template instances. Without template support it is not possible to deal with real C++ sources because even the main header files (like `iostream`) of the Standard Library contain template instances.

5. Summary

This paper gives an introduction about a project which extends GCC with *Columbus C++ ANalyzer* as a revere engineering *front end*. After a short introduction there is an overview about the construction of GCC and Columbus with focus on the *intermediate representation languages* to explain how it is possible to find a connection between the two projects and why it is useful for later optimization purposes. By briefly analyzing the theoretical background of the extension there is an overview about the implementation and technical details about the conversion of *Columbus Schema* into the GCC *abstract syntax tree*. Using this "extended compiler" I compiled C projects and C++ source files for verification of the implementation and for evolving the conclusions for further possibilities.

Thanks to this extension the capabilities of GCC were extended with new optimization possibilities, and as it is a first try for extending the compiler with a new front end, a future plan might be to generalize the development to add a module to GCC which opens the possibility of easily linking it with other, quality front end applications like EDG and others.

Acknowledgements. I would like to thank my supervisor and my advisor at Department of Software Engineering of University of Szeged, Árpád Beszédes and Gábor Lóki for their assistance in guiding me throughout this project.

References

[1] Front End Art Ltd. Homepage, `http://www.frontendart.com`

[2] GNU Compiler Collection GCC Internals, `http://gcc.gnu.org/onlinedocs/gccint`

[3] GNU Compiler Collection Homepage, `http://gcc.gnu.org/`

[4] Principles, Patterns, and Practices of Agile Software Development, The Visitor Family of Design Patterns, Prentice Hall.

[5] Department of Software Engineering, University of Szeged, GCC Code-Size Benchmark Environment (CSiBE), `http://www.csibe.org/`

[6] Ferenc, R., Beszédes, Á., Gyimóthy, T., Data Exchange with the Columbus Schema for C++, *Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR 2002)*, (Mar. 2002), 59–66.

[7] Ferenc, R., Beszédes, Á., Magyar, F., Gyimóthy, T., A short introduction to Columbus/CAN, University of Szeged, (2001).

[8] Hubička, J., The GCC call graph module, a framework for interprocedural optimization, *Proceedings of the 2004 GCC Developers' Summit*, Ottawa, Canada, (Jun. 2004), 65–75.

[9] Siket, I., Rudolf, F., Calculating Metrics from Large C++ Programs, *Proceedings of the 6th International Conference on Applied Informatics (ICAI2004)*, Eger, Hungary, (Jan. 2004), 319–328.

Csaba Nagy
Department of Software Engineering
University of Szeged
H-6720 Szeged, Dugonics tér 13
Hungary