CONSTRUCTION OF GNU COMPILER COLLECTION FRONT END

Petr Machata
Master Degree Programme (2), FIT BUT
E-mail: xmacha31@stud.fit.vutbr.cz

Supervised by: Miloš Eysselt, Lukáš Szemla
E-mail: eysselt@fit.vutbr.cz, lukas.szemla@siemens.com

ABSTRACT

The entry barrier to the GCC development got considerably lower during the last years. With work on documentation and separation of internal modules, writing for GCC becomes accessible to wide community of industry and academia developers.
This paper provides an overview of GCC high-level work-flow, with emphasis on information necessary for front end developer.

1 INTRODUCTION

GCC can translate from variety of source languages into variety of assemblers. With one command, decorated with various flags, it is able to do preprocessing, compilation, assembly and linking. How does it do this?
In following sections, we delve into successively deeper levels of overall GCC architecture.

2 COMPILATION DRIVER AND COMPILER PROPER

On the outermost level, GCC is divided into a *compilation driver*, and a *compiler proper*. Besides this, GCC uses assembler and a linker from the host tool chain.
Compilation driver is a user-interfacing application. It knows about all languages that GCC supports. Given a source file, it can guess what to do based on suffix of the file. After optional preprocessing, it launches compiler proper, then passes its output to assembler, and, eventually, linker. Depending on command line switches in effect the process can be cut short in any of the stages.
There is one compiler proper for each language, each in own executable. You can run the compiler by hand, if you so wish, it accepts much the same command line arguments as `gcc` command, and turns the source file into assembly.
Let us now look at the compiler proper closer in next section.

3 FRONT END, MIDDLE END, AND BACK END

The compiler proper itself is composed from three components: a *front end*, a *back end*, and a *middle end*. Front end contains language-processing logic, and together with middle end it makes up platform independent part of the compiler. Back end is then the platform dependent part.

Just like the compilation process done by the driver, the compilation of a source file can be viewed as a pipeline that converts one program representation into another. Source code enters the front end and flows through the pipeline, being converted at each stage into successively lower-level representation forms until final code generation in the form of assembly code [4].

There are two intermediate languages that are used on the interfaces between the three "ends" of GCC. The higher level one, used between front end and middle end, is called *GENERIC*. The lower level one, used between middle end and back end, is called RTL, or Register Transfer Language. Both middle end and back end do various optimizations on their intermediate representation before they turn it into yet lower level one.

Both interfaces mentioned are *unidirectional*: front end feeds GENERIC into middle end, middle end feeds RTL into back end. But sometimes the other direction is also necessary. For example, during alias analysis, middle end has to know whether two objects of different data types may occupy the same memory location [4]. Each language has its own rules for that, and front end is the place where language-dependent things happen. For this purpose, GCC has a mechanism of *language hooks* or *langhooks*, which provide a way to involve front end in lower layers of compilation process.

The goal of front end is to analyze source program, and ensure all types are correct and all constraints required by the language definition hold. If everything is sound, it has to provide GENERIC representation of the program. You need to know GENERIC to write a front end, but you do not have to know anything about RTL.

4 GENERIC AND GIMPLE

The important intermediate form is called GENERIC. From expressiveness point of view, it is similar to C. From notation point of view, it is similar to Lisp. GENERIC is capable of representing whole functions, i.e. it supports everything there is to represent in a typical C function: variables, loops, conditionals, function calls, etc.

GENERIC is a tree language (hence the Lisp qualities). As any well behaving tree, it is recursive in nature, having both internal and leaf nodes, with internal nodes capable of holding other internal nodes. Typical leaves are identifier references, integer numbers, etc. Internal nodes are then unary or binary operations, block containers, etc.

For optimization purposes, GENERIC is still too high level a representation. During a course of compilation, it is lowered. The intermediate code that it is lowered into is called *GIMPLE*. The process of lowering is thus inevitably called *gimplification*. GIMPLE is a subset of GENERIC. Nesting structures are still represented as containers in GIMPLE, but all expressions are broken down to three address code, using temporaries to store intermediate results[3]. There are actually two GIMPLE forms: high GIMPLE and low GIMPLE. In low GIMPLE containers are further transformed into `goto`s and labels[4].

Apart from predefined GENERIC nodes, GCC provides a mechanism to define your own node types. You have to provide a langhook for the purpose of gimplifying these. For example C++ front end actually does not use pure GENERIC, but extends it with its own node types.

5 FRONTED ASTS

While it is possible to use GENERIC for representation of programs in your front end, it is recommended not to do so [4] [5]. Your own AST representation can suit the language in hand better, and furthermore you are better shielded from the changes in GCC core. Besides, the language analysis tools that you write are then shielded from *GCC itself*, which makes them reusable in other tasks: e.g. as a syntax checker in smart editor.
This approach was taken by the Java front end, and also the experimental front end of mine, which compiles Algol.

6 GARBAGE COLLECTOR

Internally, GCC uses garbage collector [1] for its memory management. The objects with indeterminable lifetime, which includes trees, are not managed explicitly, but instead garbage-collected. The collector used is of mark & sweep kind. Pointers (variables, fields, ...) that should be collected are explicitly tagged, the tags are gathered during the build, and marking and scanning routines are generated.
The garbage collector data are also used for implementation of precompiled headers. The precompiled header mechanism can only save static variables if they are scalar. Complex data structures must be allocated in garbage-collected memory.

7 SUMMARY

In my thesis[2], I am describing the methodology of custom front end integration. The work is half exploratory, and half synthesizing in nature: there is some documentation and several papers, but in the end, few people know how to write for GCC. The thesis will provide useful how-to for anyone who wishes to write their own GCC front end.

REFERENCES

[1] Free Software Foundation. Gcc internals manual. This electronic document is available online at http://gcc.gnu.org/onlinedocs/gccint/.

[2] Petr Machata. Construction of gnu compiler collection front end. Master's thesis, Faculty of Information Technologies or the Brno University of Technology, 2007.

[3] Jason Merill. Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers' Summit*, pages 171–180, May 2003.

[4] Diego Novillo. Gcc–an architectural overview, current status, and future directions. In *Proceedings of the 2006 Linux Symposium, Volume Two*, pages 185–200, 2006.

[5] Tom Tromey. Writing a gcc front end. *Linux J.*, 2005(133):5, 2005.