

LLVMC - A Generic Compiler Driver for the LLVM Project

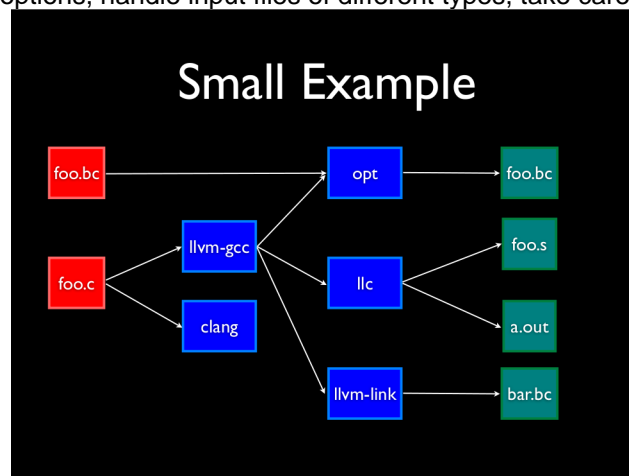
Author: Mikhail Glushenkov <foldr@codedegers.com>
Copyright: © 2008-2009, Codedgers, Inc. All rights reserved.

Abstract

This paper describes a generic compiler driver, LLVMC, which essentially plays the same role for LLVM as the `gcc` program does for GCC, but is designed to be customizable and extensible. A simple TableGen-based [TableGen] domain-specific language used to develop LLVMC plugins is presented. The driver is a part of the LLVM project [LLVM], but can also be used in other contexts.

Introduction

A compiler driver is an auxiliary tool that serves as a (command-line) user interface to the compilation infrastructure. Its job is to perform a number of transformations on the input files (usually program source code) according to some internal logic. The driver does not perform actual work, only delegates it (hence the name). This task can seem easy at first, but the devil, as always, is in the details - the driver should know about a plethora of options, handle input files of different types, take care of temporary files, etc.



A simple example of a possible compilation graph.

In the case of LLVM, the task is even more complex. Since LLVM is a flexible toolbox to build compilers with, it also needs a flexible tool to build compiler drivers. Users do not want to know about toolchain technicalities and should be provided with a single point of entry to the functionality. The first version of the LLVM compiler driver was too restrictive and inflexible and wasn't therefore used much [LLVMBug]. This paper describes the second version of LLVMC, which was written from scratch.

Background

There were several options to use instead of writing our own tool. We could just skip the trouble and recommend using a hand-written driver for each case. As already mentioned, this approach does not scale in case of LLVM, but can of course be used in some limited domains.

GCC has a 'specs' language [GCCSpecs] used for purposes similar to ours. Unfortunately, on a closer look it turned out to be not particularly user-friendly and lacking in features.

Another solution was to use a full-featured build system, such as SCons [SCons]. SCons has a lot more functionality than is needed for our aims, so this was obviously a huge overkill (not to mention the added dependency on Python). However, some of the SCons's design ideas were borrowed for LLVMC.

To sum up, the LLVM project required a driver that was:

- Easily configurable
- Self-contained (no dependencies on anything outside LLVM)
- Re-configurable at runtime
- Powerful and flexible

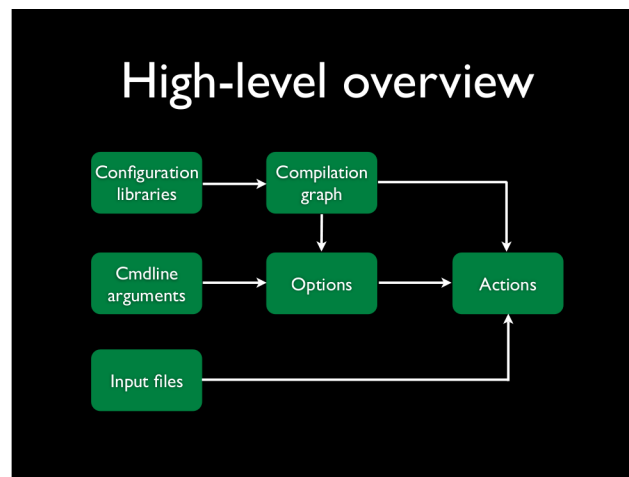
Compiling with LLVMC

The default version of LLVMC has a built-in plugin that uses the `llvm-gcc` toolchain. In general, LLVMC tries to be command-line compatible with `gcc` as much as possible, so most of the familiar options work:

```
$ llvmc2 -O3 -Wall hello.cpp
$ ./a.out
hello
```

This will invoke `llvm-g++` under the hood (the `-v` option allows to see exactly which commands are executed). For further help on command-line LLVMC usage, refer to the `llvmc --help` output.

High-level overview



A high-level overview of the system.

As already mentioned, LLVMC's job is essentially to transform a set of input files into a set of targets depending on configuration rules and user options. What makes LLVMC different is that these transformation rules are completely customizable - in fact, LLVMC knows nothing about the specifics of transformation (even the command-line options are generally not hard-coded) and regards the transformation structure as an abstract graph. Nodes in this graph represent tools, and an edge between two nodes marks a transformation path (for example, assembly source produced by the compiler can be transformed into executable code by an assembler). Since there can be several outgoing edges from a node, the edges are weighted. When passing an input file through the graph, LLVMC picks the edge with the maximum weight. The weight calculation mechanism is somewhat complicated and is described in the next section. Usually, a tool is a "one-to-one" transformation (i.e., it transforms one input file to one output file); this is not true for linkers. "Linker-like" tools therefore receive special handling and are called join tools (since they join several files into one).

The structure of the compilation graph is completely determined by plugins (also called "configuration libraries"), which can be either statically or dynamically linked. The plugin-oriented approach makes it possible to easily adapt LLVMC for other purposes - for example, as a build tool for game resources.

Writing LLVMC plugins

In this section we give a short tutorial on how to write LLVMC plugins. This is not a complete reference; LLVMC documentation (directory `doc` in the LLVMC source tree [[LLVMCSource](#)]) contains more detailed information.

A simple plugin

LLVMC plugins are written mostly using TableGen [TableGen], so the plugin author needs to be familiar with it to get anything done. The simplest possible example of a LLVMC plugin (`plugins/Simple` in the LLVMC source tree [LLVMCSource]) looks like this:

```
include "llvm/CompilerDriver/Common.td"

def gcc : Tool<
  [(in_language "c"),
   (out_language "executable"),
   (output_suffix "out"),
   (cmd_line "gcc $INFILE -o $OUTFILE"),
   (sink)
  ]>;

def LanguageMap : LanguageMap<[LangToSuffixes<"c", ["c"]>>>;

def CompilationGraph : CompilationGraph<[Edge<"root", "gcc">>>;
```

This code is then used to generate a C++ source file (using the `tblgen -gen-llvmc` command) from which a dynamic or static native library can be built. There are some build system-specific intricacies involved; please refer to the LLVMC reference documentation for the up-to-date details on how to compile plugins.

This example plugin is just a thin wrapper for `gcc`, but it lets us to illustrate some important concepts. First of all, every plugin source file should include some common definitions (the `include ".../Common.td"` line).

Plugins usually include several tool definitions, which serve as nodes in the compilation graph; the only tool definition in this example is `gcc`. Tool definitions are just lists of S-expressions (or DAGs in TableGen parlance) that specify tool properties. Most properties in the example should be self-evident; the zero-argument `sink` property means that all unknown command-line options should be forwarded to this tool.

The only obligatory plugin element is the compilation graph, which specifies the transformation structure. In the example above the compilation graph has only one edge between the `gcc` node and a special node named `root`, which marks the start link of every transformation sequence. The graph is represented as a list of edges, which can be either default or optional. Optional edges have zero weight by default, but the weight can be increased by using the conditional evaluation feature (more on this below). There can be only one default edge between any two nodes.

The last major plugin element is the so-called language map which defines mappings from file extensions to language names (so that LLVMC can choose the proper transformation sequence for a given input file). Including the language map is optional.

Options

Since one of the main tasks of LLVMC is command-line option handling, it has a relatively rich vocabulary for describing options. Several types of command-line options are recognized:

- `switch_option` - a simple boolean switch, for example `-time`.
- `parameter_option` - option that takes an argument, for example `-std=c99`;
- `parameter_list_option` - same as the above, but more than one occurrence of the option is allowed.
- `prefix_option` - same as the `parameter_option`, but the option name and parameter value are not separated.
- `prefix_list_option` - same as the above, but more than one occurrence of the option is allowed; example: `-lm -lpthread`.

- `alias_option` - a special option type for creating aliases. Unlike other option types, aliases are not allowed to have any properties besides the aliased option name. Usage example: `(alias_option "preprocess", "E")`

Options can also have their own properties, such as `help` or `stop_compilation` (see manual for details). Options can be specified either as tool properties or in an option list, which is described in the next section.

Option list - specifying all options in a single place

It can be handy to have all information about options gathered in a single place to provide an overview. This can be achieved by using an `OptionList`:

```
def Options : OptionList<[
  (switch_option "E", (help "Help string")),
  (alias_option "quiet", "q")
  ...
]>;
```

The option list is also a good place to specify aliases. Tool-specific option properties have no meaning in the context of `OptionList`, so the only properties allowed there are `help` and `required`.

Hooks and Environment Variables

Normally, LLVMC executes programs found in the system path. Sometimes, this is not sufficient: for example, we may want to specify tool names in the configuration file. This can be achieved via the mechanism of hooks that can be used from the `cmd_line` tool property:

```
(cmd_line "$CALL(MyHook)/path/to/file -o $CALL(AnotherHook)")
```

The generated code assumes that functions `hooks::MyHook()` and `hooks::AnotherHook` exist. It is also possible to use environment variables in the same manner:

```
(cmd_line "$ENV(VAR1)/path/to/file -o $ENV(VAR2)")
```

Conditional Evaluation

The 'case' construct can be used to calculate weights of the optional edges and to choose between several alternative command line strings in the `cmd_line` tool property. It is designed after the similarly-named construct in functional languages and takes the form `(case (test_1), statement_1, (test_2), statement_2, ... (test_N), statement_N)`. The statements are evaluated only if the corresponding tests evaluate to true.

Examples:

```
// Increases edge weight by 5 if "-A" is provided on the
// command-line, and by 5 more if "-B" is also provided.
(case
  (switch_on "A"), (inc_weight 5),
  (switch_on "B"), (inc_weight 5))

// Evaluates to "cmdline1" if option "-A" is provided on the
// command line, otherwise to "cmdline2"
(case
  (switch_on "A"), "cmdline1",
  (switch_on "B"), "cmdline2",
  (default), "cmdline3")
```

There are several possible tests that can be used in the `case` expression. Some examples are: `switch_on` (returns true if a given command-line switch is provided by the user), `parameter_equals` (returns true if a command-line parameter equals a given value), `input_languages_contain` (returns true if a given language belongs to the current input language set). Tests can be also combined with the familiar logical predicates `or` and `and`. The full list of test functions can be found in the reference documentation.

Debugging

When writing LLVMC plugins, it can be useful to get a visual view of the resulting compilation graph. This can be achieved via the command line option `--view-graph`. This command assumes that Graphviz [Graphviz] and Ghostview [Ghostview] are installed. There is also a `--dump-graph` option that creates a Graphviz source file (`compilation-graph.dot`) in the current directory.

Future Work

The work on LLVMC continues. Some of the planned features include:

- A `--check-graph` option that checks the compilation graph for common errors such as cycles and multiple default edges (useful for debugging).
- Support for compiling LLVMC plugins with LLVMC itself (so that TableGen-based plugins could be loaded directly).
- Support for some tricky corner cases, such as multiarchitecture binaries on Mac OS X.
- Converting existing tools in the LLVM tree to use LLVMC.

References

LLVM	The LLVM Compiler Infrastructure Project http://llvm.org
GCSSpecs	GCC 'Specs' language http://gcc.gnu.org/onlinedocs/gcc-4.3.2/gcc/Spec-Files.html
SCons	SCons: A software construction tool http://www.scons.org/
TableGen(1,2)	TableGen Fundamentals http://llvm.cs.uiuc.edu/docs/TableGenFundamentals.html
LLVMBug	LLVM Bug #686 http://llvm.org/bugs/show_bug.cgi?id=686
Graphviz	Graphviz http://www.graphviz.org/
Ghostview	Ghostview http://pages.cs.wisc.edu/~ghost/
LLVMCSource(1,2)	LLVMC Source code http://llvm.org/viewvc/llvm-project/llvm/trunk/tools/llvmc2/