How to write an LLVM optimization pass

Albert Mikó (xmikoa00), Štefan Martiček (xmarti62) @stud.fit.vutbr.cz

November 6, 2015

The LLVM framework is an open source, easy-to-extend, modular compiler framework. It offers a wide array of frontends and backends together with a powerful optimizer. LLVM is probably the best choice when you want to write a production-quality compiler for a new language, a new target architecture, or just to try out new optimization techniques. For example when writing a new language, by implementing a frontend outputting LLVM IR, you can leverage the optimizator and the existing backends.

Introducing new optimizations to LLVM is made simple by several factors. First, the LLVM Intermediate Representation (LLVM IR), a powerful language capable of storing all information about the compiled program without additional data structures. Second, the Pass Manager allowing to add user-written passes. Third, all opt passes can access the results of various analysis passes, such as alias analysis and others.

LLVM IR is a strongly typed intermediate language using SSA (Static Single Assignment) form [1]. It has 3 equally powerful representations: textual form, bitcode and in-memory. The textual form is human-readable with its primary objective being to aid debugging. The bitcode representation is a memory efficient form for longer-term storage and transfer between different tools. For example when compiling for multiple targets, the frontend and optimizer are run only once, producing a bitcode file which all the backends receive.

As writers of optimizations we are more interested in the in-memory representation, which is defined by a class hierarchy. The optimization passes interact with the code using these classes. There are two tricky instructions in LLVM IR: GEP and PHI.

The Pass Manager controls which optimization passes are called in what order. It keeps track of the validity of the analysis passes and reruns invalidated analyses when needed by another pass. Each pass declares lists of analyses whose result it requires and whose results it preserves.

All passes are divided by the scope which they work at. Basic block passes work at the lowest level, and can only influence instructions in a single basic block. Function passes are allowed to delete or insert basic blocks in a single function. Module passes work on whole translation units, and can analyse and change all functions in any order, or add/remove functions at will. New passes should always use the most restricting category possible, to enable better performance. [2]

User-written passes can be compiled into a dynamically linked library (.so or .dll) and run via `opt --load=MyPass.so --my-pass`. Another possibility is writing a custom driver where more passes (user-written or core LLVM) can be run in a predefined order.

In our talk, we describe the process of creating and running a custom optimization pass using a simple example.

References

[1] http://llvm.org/docs/WritingAnLLVMPass.html *Writing an LLVM Pass.*

[2] http://llvm.org/docs/LangRef.html *LLVM IR Language Reference.*