Undecided

Ashutosh Rishi Ranjan

May 9, 2016

Chapter 1

Plan

Chapter 2

Abstract

Intermediate languages in compilers glue two different compiler construction phases. Being machine independent makes it possible for an IR structure to transform from one form to another to allow different patterns of reasoning, and different stages of transformation. In this thesis the proposal of a logical language based IR is explored by inserting it in a incremental compiler pipeline as a mid-level IR. This IR lies in between the source code and the LLVM IR, enabling Logical programming analysis techniques to be used for optimisation before the usual LLVM optimisation. Using this compilation pipeline we build an incremental compiler which is focused on LP analyses and reusing IR structures already built in previous instances compilation of a target.

Chapter 3

Introduction

This thesis explores using a logical intermediate representation (IR) in a incremental compiler pipeline for a new declarative and imperative mixed paradigm source language. This logical IR is called LPVM, and its simple clausal structure enables logical programming analysis and easier optimisations. It lies middle of the compilation pipeline, before an LLVM generation stage, making it a middle level IR. Using this representation, we can also make the compiler incremental and have a lazy build system which tries to avoid re-compilation as much as it can. Even though having multiple IR stages and forms adds extra work in the compilation process (and compiler construction), we show a simple transformation of LPVM to LLVM, so that the compiler is able to target all the machines LLVM can realistically.

The source language is called Wybe. It's a new language which aims to unify the good parts of declarative and imperative languages. Having a mixture of paradigms makes it a good fit for a compilation pipeline which involves a mixture of paradigms as well.

The data structure of an Intermediate representation is an important factor in deciding what optimisation passes are going to be useful. Different compilers usually have their own IR, which maybe only slightly different from other IRs. The actual form is really a compiler construction choice. There are efforts to build a universal IR, like the LLVM project, but a reasonably complex language would have it's own unique requirements which can't be accounted for in a single universal IR or form. The IR generation stage in a compiler pipeline goes through multiple optimisation passes. There is no restriction on the form of the IR as it moves through these passes. In fact having multiple IR forms, which gradually transform from being closer to the source language to a more machine dependent form is quite common. Multiple forms opens up multiple approaches to optimisations.

Mid level IRs, quit simply lie in between some high level IR or source code and a low level IR. We insert LPVM into the compilation pipeline as a mid level IR between the source code and the LLVM IR. In a way, our target code is the LLVM IR. Even though LLVM code generation comes with it's own set of curated and tested optimisations, its main use here is to avoid the need to account for every architecture. Thus we can focus on maximising the usefulness of LPVM. We also show how the clausal form of LPVM can be easily translated to the more imperative block style of LLVM.

The Wybe compiler wants to be lazy and incremental. It wants to avoid as much recompilation as it can. The object files Wybe compiler builds have enough information to be used in place of a source file, while at the same time provide inalienable code which could only have been obtained from the source code. The simple clausal semantics of LPVM is the key to this. To be more incremental, the compiler maps and tracks the source code semantic structures to LPVM clauses. For re-compilation, it tries to load the already compiled clauses stored in the object files for any trivial changes of the source.

Another focus of the Wybe compiler is to have a build system ingrained into the compiler. We try to mimic the gnu make utility with some simplifications, and hence our Wybe compiler is usually run through the command 'wybemk' (Wybe make). Instead of having a separate make file, the Wybemk command just takes a target name, infers the dependency chains and the list of files to compile and link and makes the target. We wanted to have the ability to link in foreign source object files and not just Wybe source files.

Chapter 4

Literature Review

4.1 Horn Clauses as an Intermediate Representation

This paper outlines the structure and usefulness of a logical language IR. An implmentation of this form, called the LPVM, is also presented. This is the IR used in the Wybe compiler, before the LLVM generation stage.

Chapter 5

Wybemk Build System

Wybemk is the wybe compiler and wybe make system joined together in one utility. The wybemk compiler just needs the name of a target to build, and it will decide the building and linking order. Targets include architecture object files, LLVM bitcode files, or a final linked executable.

Similar to the gnu make utility, it does not want to rebuild any object file which is already built. Taking it one step further, the object files wybemk makes store useful LPVM information generated from the source code. While a normal object file does present it's name table, allowing the functions stored in it to be called as extern functions, these functions are not inlineable. While source code for functions can be stored in a section of the object file, it is easier to just pass the original source file around with the object file. The LPVM structure of a parsed source file however is simpler in its structure. A body of a function is simply a list of clauses. Storing this information in the object file enables inlining in at the LPVM generation stage. Inlining also presents a wider view for analysis as we can look at the body of the functions we want to call externally.

Storing a serialised form of LPVM in the object file is quite similar to the LLVM bitcode files, which stores the LLVM IR in a byte file. To keep this option open, since we use LLVM as the final stage, we also present a wrapped LLVM bitcode structure which stores both the LLVM and LPVM IR in it. Such a file can be linked in by wybemk or by the LLVM compiler, which loads different information from the file.

5.1 Storing structures in Object files

5.2 LPVM as a stored structure

5.3 Loading Inlinable LPVM