Undecided

Ashutosh Rishi Ranjan

May 26, 2016

Chapter 1

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 2

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.

Why use a logical IR?

Chapter 3

Literature Review

Chapter 4

LPVM

4.1 Horn Clauses as an Intermediate Representation

[Gange et al., 2015]

The paper describes a new form of IR using a logical programming structure, now called LPVM. The wybe compiler presents an implementation of this clausal form and as such LPVM is a core part of its pipeline. This thesis provides the final code generation stage for LPVM and wybe.

LPVM is an attempt to make program analysis and transformation easier. It's an alternative to the common imperative IR forms while maintaining a similar set of program analysis methods. The paper compares LPVM against the other current options for an IR, like the three address code and SSA, discussing the drawbacks of these common IRs. The focus is on solving these drawbacks at a more fundamental level rather than relying on further transformations. The motivation outlined in the paper is having a simple limited structure in analysis, while drawing in existing logical programming reasoning. The limited set of primitives, additionally enables the wybe compiler to be more efficient by embedding a LPVM transformation in the object file. This would have been inefficient with a more complex structure.

The Three Address Code (TAC) IR and its refinement, the Static Single Assignment (SSA) form, are popular IRs used in compiler constructions. They are simple enough to be universal and are quite extensible to accommodate different source language semantics due to their little restrictions. A SSA based IR will generally be laid out as basic blocks and branching instructions connecting them, like a graph. The SSA refinement requires all variable names to be unique in a block. This makes it easier to track variable lifelines. But this also requires a virtual function called phi to choose the value of the same variable coming in from two different blocks. The paper points out that this is inadequate for determining the control flow path. While the phi function can tell us which blocks are providing the values for the same variable during analysis, it won't tell us the condition that was evaluated for that decision. This can be done by looking into all the predecessor blocks that provided the value. Another solution to this, as pointed out by the paper, is using the Gated Single-Assignment (GSA) form which augments the existing phi function to capture the block entering condition.

Another drawback described is the SSA form being biased to forward analysis. The phi function at the top of a block is helpful for analysing that block locally. A predecessor block (to the block with phi) just knows about its branching destination, and not about the phi function choosing between its variables and other blocks. A transformation to the standard SSA form, called the Static Single Information (SSI), provides a solution to this problem by adding another virtual function to the end of the blocks which describes the variables that will be compare with other blocks in the subsequent block.

So far every drawback of SSA has been addressed can be solved by creating an extension to the SSA structure. While these are perfectly feasible, they are additional complexities. In the case of LPVM these problems are solved at the basic structural level, without any special functions. During analysis, the special functions of SSA have to be resolved with extra work. An analysis of a block containing phi node involves revisiting the analysis work done for the predecessor blocks. This is a result of SSA's naming rules. Even though SSA enforces unique names for every declaration, alternate blocks can still have the same variable names. A more functional translation of SSA avoids this, but still exhibits the forward bias problem.

The LPVM IR is a restricted form of a logical language. It does not feature non-determinism seen commonly in a LP. Therefore all input parameters have to be bound to a value before calling that procedure. It also requires fixing the mode of a parameter. In LP, a procedure can have a parameter which can behave as an input or an output. LPVM requires this behaviour to the explicit and fixed for every parameter for easier reasoning. These restrictions makes LPVM surprisingly easy to read.

At the top level LPVM form there are only procedures. In the form presented in the paper a procedure consists multiple *Horn Clauses*. A horn clause with its *Head* describing its parameters and procedure name, and the sequence of goals as the body can be seen as a procedure itself. Since LPVM wants to be deterministic, only one of the clauses with the same name evaluated for given inputs. That clause can be seen as the entire procedure itself with the *Head* as the procedure signature, without bothering about the usual backtracking in LP. The parameters to a clause also needs its modes to be explicitly defined: either as an input or an output. There may be two alternative *Clauses* of the same name having switched up modes for the same parameters. Since determinism will select only one of them as the procedure, single modedness is preserved.

The *Goals* are simple body statements. Logical statements in SSA result

$x0 = x1 + x3$	$wybe.int. + (x1, x3, ?x0)$
$if\ (x0 > 0)\ left\ right$	$wybe.int. > (x0,\ 0,\ ?tmp1)$
	$case\ tmp1\ of$
	$0:\ left..$
	$1:\ right..$

Figure 4.1: SSA statement and their equivalent LPVM goals

$$\begin{aligned}
Proc &\rightarrow Clause* \\
Clause &\rightarrow Proto\ Body \\
Proto &\rightarrow Name\ Param* \\
Param &\rightarrow Name\ Type\ Flow \\
Body &\rightarrow Prim*\ Fork \\
Prim &\rightarrow PrimCall \mid PrimForeign \\
Fork &\rightarrow Var\ Body*
\end{aligned}$$

Figure 4.2: LPVM Implementation Data Type

in block cause branching. These statements are represented as guard goals in LPVM. When a guard goal is encountered, a new clause will be created with the same goal sequence until that guard, but will follow the complimentary evaluation of the guard after that. In the implementation however, the procedure body or the clause body or the sequence of goals are put in a tree data structure. Instead of having somewhat duplicated sequence of goals in creating an alternate clause, the guard goals instead create branching child nodes, each with further sequence of goals. This is similar to the layout of basic blocks within a function in SSA, which makes it easier to translate the LPVM to any SSA based IR later for code generation.

At the end of conversion to LPVM, the IR form only contains procedures with multiple clauses. Alternate branches of conditionals are incorporated in their own clauses, chosen by the evaluation of a guard statement. Unconditional jumps are procedural calls and loops are flattened to recursive procedure calls. Every procedure will also define all its input and output parameters explicitly in its signature. Since output variables are specified here, there is no need of a return instruction. The only variables that would exist in the clause bodies are now the specified input and output variables, and temporaries between them. This declarative paradigm greatly simplifies a lot of analyses. This also helps determine the purity of a procedure.

4.2 Current Implementation of LPVM

The Figure 4.2 shows the algebraic data type used to hold the LPVM IR. This representation has wybe sensitive information like wybe types and module definitions stripped away for easier discussion. The actual implementation will be looked at again in a latter chapter.

In the LPVM implementation used by the Wybe compiler every goal is a primitive procedure call. At the most basic level these are similar to primitive machine instructions. This again simplifies code generation. As mentioned above, instead of having duplication of clause bodies the guard goals currently result in a fork of bodies. Generation of two complementary clauses similar to the LPVM specifications in the paper is trivial from this tree structure.

All loops and conditionals are un-branched by generating new procedures which are involved in recursion. The calls are tail calls and hence the code generation will have to ensure tail call optimisation is enabled. Simple conditionals like if conditions, just generate a fork in the body. The alternate destinations, say *Body A* and *Body B*, are stored in a list in the *Fork*. Since the *Fork* is at the end of a clause *Body*, the remaining body (after the If statement) is placed in a new generated procedure. This generated procedure is called at the end of *Body A* and *Body B*.

Chapter 5

Wybe programming Language

Wybe is a new multi-paradigm programming language, featuring both imperative and declarative constructs. At the top level it contains both functions and procedures. A function header specifies the inputs and their types, and the output expression type. The body of the function will be an expression evaluating to a value of that specified out type. Whereas a procedure header species its inputs and outputs (along with their types), and any mutable or external resource it works with (like IO). Its body will contain sequential statements which build those outputs from the inputs. There is no return statement, as at the end of the procedure body the specified outputs will be returned. In a way, a procedure header lists the parameters which will be used in its body, and fixes the flow mode (input flow or output flow) for each of them.

Wybe is statically typed with a strong preference for interface integrity, for which a function or procedure header should define all it's input and output types, along with any mutable resources that will be affected. By forcing the information flow to be explicit, Wybe makes it easy to determine the purity of the function just by looking at a function or procedure prototype. There is also no requirement for an interface or header file.

With Wybe, a module is equivalent to a wybe source file. The module name is same as the source file without the extension. The module's interface consists of the public functions and procedures in the module. There is also a separate syntax to declare sub-modules inside the full file module. Sub-module names are qualified with the outer modules' name. For example a module *A.B.C* is the sub-module of *A.B*, which is a sub-module of

$$\begin{array}{rcl} func\ factorial(n:int):int & \rightarrow & proc\ factorial(n:wybe.int, ?\$:wybe.int) \\ ?c = bar(a,b) & \rightarrow & bar(a,b,?c) \\ ?y = f(g(x)) & \rightarrow & g(x,?temp)\ f(temp,?y) \end{array}$$

Figure 5.1: Normalisation of Wybe functions to Procedures.

```
public type int is i64

    public func +(x: int, y: int) : int = foreign llvm add(x,y)
    public proc +(?x: int, y: int, z: int) ?x = foreign llvm sub(z,y) end
    public proc +(x: int, ?y: int, z: int) ?y = foreign llvm sub(z,x) end
    public func =(x: int, y: int) : bool = foreign llvm icmp eq(x,y)

end
```

Figure 5.2: Sample of the wybe.int module from Wybe standard library

```
public type bool = public false | true

    public func =(x: bool, y: bool) : bool = foreign llvm icmp eq(x,y)
    public func /=(x: bool, y: bool) : bool = foreign llvm icmp ne(x,y)

end
```

Figure 5.3: Bool type as an Algebraic Data Type from the Wybe standard library

A, which is the module of the source file *A.wybe*. Defining new types also creates a new sub-module. All dependencies of a module can be inferred through the top level *use module* statements in the source file.

Types in Wybe are simply modules. Standard Wybe considers int, *float*, *char*, *string* as primitive types. In reality these are just types provided by the Wybe standard library module, which can be replaced with any other flavour of a standard library. Defining basic types requires just specifying its memory layout (in terms of word sizes) and providing procedures or functions to interact with the basic type. A sample from the *wybe.int* type module is shown in Figure 5.2. This type definition is in the source file *wybe.wybe*, making *int* a sub-module of the module *wybe*. This *int* type sets the size of its members to be *i64*, a syntax borrowed from LLVM, occupying 64 bits.

More compound types can be defined in terms of basic types or other compound types. Wybe has *algebraic data types*. The *bool* type can in this way with two type constructors, as shown in Figure 5.3. The compiler will infer the storage for members of this type.

Procedures (and even functions) in Wybe can be polymorphic. Multiple Wybe procedures can have the same name but with different paramter types and modes. For example, a procedure to add two numbers can have the following prototypes: *add(x, y, ?z)*, *add(x, ?y, z)*, *add(?x, y, z)*. The call *add(3, ?t, 5)* will evaluate *t* to be 2. This selection is done in the Type checking pass during compilation, which matches calls to the definition with the correct types and modes.

5.1 Compilation of Wybe to LPVM

The Wybe syntax tree is slowly transformed to the LPVM IR structure. In this process it undergoes *flattening*, *type checking*, *un-branching*, and a final clause generation pass to obtain a structure similar to Figure 4.2. Variables in Wybe can be adorned to explicitly define their direction of information flow. A variable can flow in (x), flow out (?x), or both ways (!x). The Wybe model of explicit information flow is quite similar to the LPVM predicates, making LPVM a good fit for this language.

5.1.1 Flattening Pass

During compilation everything is converted to a procedure quite early in the pipeline. The functions and expressions are normalised to look like a procedure definition along with the flattening step by the compiler. The output of the function is simply added as a out flowing parameter in its procedure form. Expressions are dealt with in a similar way. Some common conversions are shown in Figure 5.1.

Since LPVM primitives are in the form of procedure calls, all normalised Wybe statements are gradually reduced to procedure calls too. These primitive procedure calls can be calls to other procedures in the module or imported modules (fully qualified procedure names), or be foreign calls. Foreign calls reference procedures or instructions which have to be addressed later by linking in some library which provides it. For example, the wybe standard library defines *println* whose body statements are foreign calls to C's *printf*. A shared C library will be linked with the standard library to resolve these calls to access system IO later. To Wybe and LPVM the only difference between a local and a foreign procedure call is that the local calls can be in-lined since their definitions will have a LPVM form in another wybe module. Otherwise it is just another *Prim* (primitive) in a LPVM clause body.

5.1.2 Type Checking Pass

Wybe is statically typed, so having a type checking pass is essential. Every variable name in the AST will be annotated with an inferred type. This pass connects type names to the modules that provide the definition for that type. This is required as even standard types like *int* can be provided by a non standard library just as easily. Polymorphic calls are resolved to the actual definitions here.

Type definitions include functions and procedures which work with the defined type. For example, the equality function '=', can be defined in a type module for *int* and the type module for *string*. The type checker will choose one depending on the context. A statement comparing two *int* (inferred) variables the call *proc call =(a, b, ?c)*, will be converted to *proc call wybe.int.=(a:wybe.int, b:wybe.int, ?c:wybe.int)*. By the end of a successful type checking, every flattened procedure call and variable types names will have an annotation of the fully qualified module that defines it.

5.1.3 Un-branching Pass

The un-branching pass is where all conditional branches and loops are replaced with procedure calls and recursion respectively. This is the structure defined by LPVM. At this stage, a flattened Wybe procedure may create one or more generated procedures to act as branching blocks, as described in the chapter on LPVM.

Chapter 6

Wybemk, Compiler and Build System

Wybemk is the incremental compiler and a Make utility combined together in one executable for Wybe source code. It is modelled after the GNU Make utility, but doesn't need an explicit Makefile to make Wybe source files. The Wybemk compiler just needs the name of a target to build, and it will infer the building and linking order. Targets include architecture dependant relocatable object files, LLVM bitcode files, or a final linked executable. The object files and bitcode files that Wybemk builds are a little different than what other utilities create. They have embedded information that assists the future compilation processes in being incremental. It is this embedding that allows Wybemk to be an incremental and work-saving compiler.

The elementary work saving features are very similar to what the GNU Make does. It does not want to rebuild any object file which is already built and is newer than its corresponding source file. But by doing so, the LPVM form and analysis for that module is also skipped. This is acceptable for only intra module optimisations, since the final optimised object code will be the same. But we might be missing a lot of inter module optimisation opportunities and LPVM inlining that other dependant modules can reap benefits from. Object files store a symbol table which will list all the callable function names in it. This is what the *linker* uses to resolve extern calls during linking. The body of these functions are stored in object code form. We can't make a decision on inlining these functions into another module from this. It would be nice to have the LPVM form of all the modules participating in a compilation process for these optimisation decisions. We want to store LPVM analysis information in the object files so that when they are not going to be re-compiled, we can at least pull in the LPVM form of that module in the compilation pipeline.

The limited structure of LPVM makes serialising and embedding its byte structure into a object file byte string easy and feasibile. We could have also stored the parse tree, but a parse tree has a wider form and is redundant with the source code. With storing the parse tree we are only skipping the work the parser does and would have to redo all the LPVM transformation and analysis. This would be more work. The simple yet highly informational form of LPVM makes it an ideal structure to pass around.

Why object files though? An object files' structure is architecture dependant and requires different efforts for storing and loading information for each architecture. This would put a constraint on the number of architectures that Wybemk can operate on eventhough with LLVM it should be able to possibly generate code for those architectures. However object files are a common container for relocatable machine code. Most compilers traditionally build a object file for the linker to link. Currently Wybemk does not want to reinvent that format and we would like our incrementality features to work in tandem with the common choices. Apart from object files however, the Wybemk compiler can do the same embedding with LLVM bitcode files. LLVM bitcode files can be treated as architecture neutral, and since we use a LLVM compiler as a final stage, we can use bitcode files as a replacement for architecture dependant object files too.

6.1 Storing structures in Object files

Object files store relocatable object code which is the compiled code generated by the LLVM compiler in the wybe compiler. Even though different architectures have their own specification of the object file format, they are modelled around the same basic structure. Object files defines segments, which are mapped as memory segments during linking and loading. The segment TEXT usually contains the instructions. An object file also lists the symbols defined in it, which is useful for the linker to resolve external function calls from another module being linked. Avoiding all the common segment names, it is possible to add new segments to the object file (at the correct bytes), which do not get mapped to memory. Using such such segment we can attempt storage of some useful serialised meta-data in the object file.

To emulate the behaviour of gnu make, or to avoid rebuilding an object file, it is quite trivial to check the file creation date and reach a decision. However this hinders a lot of LPVM level optimisation. Since there is no need to compile a source module if it has a newer object file for it, we can't do any inter-module analysis with it. We can get the symbol names and possibly the generated object code for the function names, but those are not useful for LPVM analyses. Storing generated LPVM code in the object files would circumvent this. The simple semantics of LPVM is an excellent candidate for storage in object files without being disruptive. Even when the source code file is present, loading the LPVM code from the object file is faster than engaging the parser and generating LPVM from the parse tree.

LLVM also generates bitcode files. These files store a serialised form of the generated LLVM IR. The LLVM compiler can transform these to the machine local object file too. Being a simple bitstream format, it is possible to include other serialised metadata into this file as long as the LLVM compiler can find its required IR. Storing LPVM in these files is a feasible alternative to the machine local object file formats. LLVM bitcode file can be used for code generation on any valid target, much like the Java bytecode.

6.2 LPVM as a stored structure

The simplicity of LPVM makes it effecient...

6.3 Loading Inlienable LPVM

Talk about object file file structures?

Bibliography

[Gange et al., 2015] Gange, G., Navas, J. A., Schachte, P., Søndergaard, H., and Stuckey, P. J. (2015). Horn clauses as an intermediate representation for program analysis and transformation. *CoRR*, abs/1507.05762.