Undecided

Ashutosh Rishi Ranjan

June 1, 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 Logic Programming analyses 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 presents an incremental and work-saving compiler which utilises a newly proposed logic programming based intermediate representation, called **LPVM**, in its pipeline. The compiler itself compiles a new multi-paradigm source language called **Wybe** (pronounced *Wee-buh*). The compiler eventually targets the **LLVM** compiler framework which allows the compiler to be flexible in targeting multiple modern targets, and provide a working implementation. Ideas explored in this thesis are stemmed from the motive of achieving efficiency in a compilation pipeline by avoiding redoing any work done before. The source language Wybe is a complete full featured language targeted at being easy to learn and being suitable for production use. While the use LPVM itself provides us plenty of benefits in program analysis and optimisation, we also exploit its structure to achieve incremental features in our compiler. This will push Wybe to be even more suitable for large scale projects.

The choice of the intermediate representation to use in the compiler is an important one. It is the data structure which abstractly holds the source code semantics, while being machine independent. It undergoes many optimisation passes which may or may not alter its structure, and can be targeted to any architecture for code generation eventually. The choice of the IR structure also makes certain program analyses easier than the rest, or even enables some which other structures might not allow. There can multiple IR forms between the source code and the machine code (Johnson, 2008). IRs which resemble source code semantics more closely are deemed to be on a higher level than the ones which have a closer resemblance to the machine code. An IR will strip away information it does not need anymore, or information that it has utilised completely, and transform to a simpler lower level form. In our pipeline we have two IRs. The **LPVM IR** form is generated from the source program in Wybe, and the **LLVM IR** form is generated from the final LPVM IR form. All the major optimisation and analysis passes happen on the LPVM form. The LLVM form is generated on a simplified version of LPVM (after optimisation) and is just needed so that we can generate machine code without duplicating effort.

LPVM IR is the implementation of the logic programming IR form given in Gange et al. (2015). The term *LPVM* is an abbreviation for *Logic Program Virtual Machine*, and named so in a similar fashion to *LLVM*. They are not really virtual machines. LPVM has an incredibly simple IR structure and makes it really easy to for reasoning with different program analysis techniques. It also allows the use of powerful logic programming analysis. In this thesis, the focus is more on exploring the structural benefits of LPVM in being incremental, rather than its proposed and possible optimisation techniques.

The Wybe language compiler is called **Wybemk** (pronounced *Wee-buh-mik*). It is a combination of *Wybe* and *Make*. To support our incremental features, we need to embed certain information on a compilation into the object files the compiler creates. These object files are the same as the architecture specific object files, but with extra docile information usable by Wybemk. 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.

The rest of the thesis presents the research and work done to build the incremental compilation pipeline for Wybe. The sections try to follow the stages of the compilation in order. In Chapter 3 we discuss the literature explored to build the compiler, including discussion on the important papers which form a base for this thesis. In Chapter 4 we present the implementation of LPVM used in the compiler. In Chapter 5, an introduction to the Wybe programming language is given, and in Chapter 6 its transformation to LPVM is discussed. Having described the structure of the source language, LPVM, and its representation in the compiler, we present our incremental and work saving build system, Wybemk, in Chapter 7. Finally we discuss code generation to LPVM in Chapter 8.

Chapter 3

Literature

The logic program based intermediate representation presented in Gange et al. (2015) is the IR used in the Wybe compiler. It is an integral stage of the compilation pipeline and its features and benefits affect a lot of decisions made in the construction of this compiler. The Wybe compiler doubles as a showcase for a working implementation of the proposed logic IR. A discussion of this paper is presented in section 3.1.

Our goal is to build an incremental compiler which exploits the structure of the LPVM IR. Examining the approaches taken by other systems for building incremental compilers and understanding their obstacles is important so that we can build our system smoothly. An incremental compiler for C++ given in Cooper and Wise (1997) presents a working system to compile at the function (or object) level instead of the file (or module) level. Their approach is discussed in section 3.2.

It's always useful to have a build system in-built in the compiler. The Wybemk compiler tries to emulate the famed GNU *Make* (Feldman, 1979) build system, but without a *Makefile*. The target is passed on the command line and the build happens only when the source file is newer than the target in its file modification time. Internally a dependency graph is generated, like *Make* and a depth first traversal is done for building dependencies.

To produce a working compiler we hook into the LLVM compiler infrastructure (Lattner, 2002) by providing a transformation from the LPVM IR to the LLVM IR. While not an universal IR, LLVM IR is an excellent target due to the amount of work that is being done on its machine code generation. Many popular languages like Haskell, Rust, etc. are getting LLVM backends for the same reason. We won't need to rely on all the optimisation possibilities of LLVM since we are reliably doing majority of our program analysis and simplifications on LPVM.

3.1 Horn Clauses as an Intermediate Representation (Gange et al., 2015)

The paper describes a new form of IR using a logic programming structure, now called LPVM in the Wybe compiler. Since this thesis extends the LPVM implementation by providing code generation for it and harnessing its features in building incremental features in the compiler, it's important to understand the reasoning behind the structure of LPVM and its benefits over its counterparts. LPVM can be compared with the commonly used IR forms like the Three Address Code and its Static Single Assignment (SSA) extension (Alpern et al., 1988). The paper presents sound discussions of the drawbacks of these forms and the other solutions used to solve these drawbacks. The other solutions listed extend the SSA form to address its limitations, whereas LPVM does not need to make an attempt at doing so. It instead presents a completely different structure that is free from these drawbacks from the get-go. This structure uses *Horn Clauses* from logic programming imposing certain limitations on that form.

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 can accommodate different source language semantics due to their fairly open structure. They can be constructed efficiently (Cytron et al., 1991) and allow numerous useful optimisation techniques. 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 *φ-function* to choose between the versions of the same variable coming in from two alternate predecessor blocks since each of those blocks will have its own name for that variable. Thus, there will be a *φ-function* for every variable whose value can arrive from alternative predecessor paths into the current block. This function is *fake* and will not have code generated for it. Instead it's evaluation is solely for program analysis and requires backtracking into analyses of the predecessor blocks to determine the actual path taken by the control and determine the resulting abstract value. Even though the SSA form is visually simple, it's construction may not be so. The limitations of this form is part of the motivation for presenting LPVM.

A *φ-function* does not provide information on the control flow path. Computation of the path taken will have to done backwards, by looking into the predecessor blocks. Another extension presented in Ottenstein et al. (1990), called the Gated Single-Assignment (GSA) form, augments the existing *φ-function* function to capture the block entering condition. This makes path determination easier, but at the cost of adding more complication to the SSA form. The LPVM form on the other hand has a more explicit information flow in its basic structure.

SSA is useful for local block analyses. But the branching of blocks and the joining of incoming variables with *φ-functions* are biased to forward analysis. In backward analysis it's not trivial to know of the alternate blocks holding alternate versions of the variables in the current block. This bias is avoided with another extension called the Static Single Information (SSI). This form includes another virtual function (σ) to the end of the blocks which branch into alternate blocks which describes the destinations of each alternate variable. LPVM provides this information easily as part of its basic structure.

The unique assignment restriction results in alternate versions of the same variable in diverging blocks. Which in turn requires extra work for converging back into one variable. Another functional programming form of SSA (Appel, 1998) is discussed which avoids duplicate versions of a variable by replacing branching with function calls and *φ-functions* with parameter passing. Every alternate block will replace its jump to the converging destination with a function call. Even though this is a declarative form just like LPVM, it still makes information flow explicit only in the forward direction by specifying only the in flowing parameters. LPVM instead provides the input and output parameters for a 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.

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. These restrictions makes LPVM surprisingly easy to read and reason with.

At the top level LPVM form there are only predicates (or procedures). In the form presented in the paper a procedure consists multiple *Horn Clauses*. The *Head* of a *Clause* describes its parameters and predicate name. These parameters can be in-flowing or out-flowing. In the abstract model the output parameters are separated from the input parameters with a semicolon. If we look at a *Clause* body as a block in SSA, then unlike SSA we have explicit information on all the variables entering and exiting the block without any extra virtual functions.

For a predicate or procedure call in LPVM, only one of its *clauses* will be executed. This is due to LPVMs' enforcement of determinism. That clause can be seen as the entire procedure itself with the *Head* as the procedure prototype. 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

$$\begin{aligned} gcd(a,b,?ret) \rightarrow\ & \textbf{guard}\ b! = 0 \\ & \wedge\ mod(a,b,?b') \\ & \wedge\ gcd(b,b',?ret) \end{aligned}$$

$$\begin{aligned} gcd(a,b,?ret) \rightarrow\ & \textbf{guard}\ b == 0 \\ & \wedge\ ret = a \end{aligned}$$

Figure 3.1: Comparison of SSA and LPVM for the gcd function.

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 in the *Clause* body can be a guard goals (conditionals) or be simple goals. Guard goals should create a fork in the control flow through the body. But in LPVM this is mitigated by creating another *Clause* which has the same sequence of goals up to this guard, but thereafter follows the complimentary evaluation. Hence, the control flow is explicit unlike SSA. The actual implementation, discussed in a later chapter, is a little different in its data structure for branching. But the behaviour is preserved.

In comparison with SSA, the basic blocks are replaced with *clauses*. The branching and jumps is replaced with procedure calls. Each procedure provides the names of the variables moving in and moving out of it, favouring both directions of analysis. Loops are replaced with recursive procedure calls. We know which variables the body of the procedure (or its *Clause*) will be building up for outputs just by looking at the procedure signature. There is no need for return instructions or *φ-functions* . This also makes purity reasoning explicit. Everything that a LPVM procedure affects (in terms of *registers* or other *resources*) have to be declared in the signature or *Head*. The Figure 3.1 demonstrates these differences of semantic structure between the SSA and LPVM for a simple *gcd* function.

3.2 Achieving Incremental Compilation through Fine-grained Builds (Cooper and Wise, 1997)

The system given by Cooper and Wise (1997) is a incremental integrated program development system for C++ called *Barbados*. It contains a build system with a granularity of functions and procedures instead of the usual file level granularity. This is quite similar to the build system we want for Wybe. The requirements for building such a system, as listed in the paper, involve automatic dependency inference, transparent compilation, and ensuring no old code is executed. These are also the requirements we want the Wybe compiler to follow. The actual implementation of the system is quite different from what we want in our compiler though. While Barbados focuses on building an interactive system which lazily compiles code given to it, while deciding whether to do a re-compilation, Wybe wants the incremental features to kick in during compilation of a full source code module. The compilation is also done from the source code to object code in Barbados, whereas for the Wybe compiler, the tracked compilation will be considering the LPVM structure in the middle too.

The code structures that Barbados considers at the lowest granular level of compilation are chosen in a way that dependencies between them can be generated automatically. The first step highlighted by the paper is the tracking of these dependencies. There is also an emphasis on having a separation of interface and body for all of the basic entities of compilation. If an interface is able to completely reflect a need for compilation in the body then the body is only re-compiled on an interface change. The use of time stamps along with the dependency tracking can ensure that the correct versions of compiled code can be used. These constraints are sound and tested, and as such are ensured in the Wybe compiler.

Barbados tackles the dependency tracking problem by maintaining a tree like structure to show dependencies. The entities can be involved in a transitive closure of dependencies. For every compilation the root of the tree is targeted. The system then moves through the tree until it reaches a leaf and then works it way back from there. This way it can ensure that is it dealing with dependencies in the correct order. There are a couple of problems that can arise with this structure such as circular dependencies and the volatility of dependencies during tree traversal. These problems are solved by multiple traversals of the tree. The paper finds that the time spent in multiple traversals is negligible when compared to compilation times, providing support for this approach. The heuristic for change in an entity is a time stamp. While these are effective for propagating change, it may be missing cases when the same entity is saved over the old one. The time stamp changes but structurally nothing has changed.

Chapter 4

Implementation of LPVM

The actual implementation of the LPVM structure differs a bit from the abstract structure proposed in Gange et al. (2015). For a guard goal, the abstract representation results in having two clauses with the same initial sequence of goals up until complimentary guard goals. In the implementation however, to avoid duplication, there is only one clause generated. The basic clause body is a sequence of goals barring a guard goal. At the guard goal a fork is created which contains the fork condition, and a list of basic subsequent clause bodies, each for an outcome of the condition. A binary condition will have two basic bodies in the fork list. This is similar to basic block branching in SSA, however the diverging branches don't need to converge in a body and hence there will be no need for *phi* like functions.

In a way LPVM procedures or predicates are polymorphic since a call to them will execute any one of the clauses under them. There can be a *Clause* for different modes of the procedure parameters. For example, the procedure $-$ can have clauses $-(a, b, ?c)$ and $-(a, ?b, c)$, marking different operand positions as outputs. Even though each of the clause functions in a single mode, a procedure can be made to exhibit multiple modes of a logic programming language. This construct is very similar to the polymorphism Wybe has to offer, and is such this construct directly enables it it.

The Figure 4.2 shows the algebraic data type used to hold the LPVM IR.

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

Figure 4.1: SSA statement and their equivalent LPVM goals

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

Figure 4.2: LPVM Implementation Data Type

```
do a
   if b: Break
   else: c
end
d
```

$$
\begin{array}{rcl}
\textbf{proc}\ \mathit{next}1 & \rightarrow & a\ \mathit{gen}1\ \textbf{end} \\
\textbf{proc}\ \mathit{gen}1 & \rightarrow & \textbf{guard}\ b\ 1 : \mathit{break}1\ \textbf{end} \\
 & \mid & \textbf{guard}\ b\ 0 : \mathit{gen}2\ \textbf{end} \\
\textbf{proc}\ \mathit{gen}2 & \rightarrow & c\ \mathit{next}1\ \textbf{end} \\
\textbf{proc}\ \mathit{break}1 & \rightarrow & d\ \textbf{end}
\end{array}
$$

Figure 4.3: LPVM Procedures generated for an Imperative Loop

This representation has Wybe sensitive information like Wybe types and module fields stripped away for easier discussion. In the implemented data type, the term *Goal* is replaced with *Prim*, for primitive. These primitive statements are meant to reflect source and target code semantics. They are just procedure calls and the only semantic information they contain are the procedure name and the signature. These can be used to refer to source language procedures or machine code (or LLVM) instructions just as easily. Local calls look like: *factorial(tmp$10: int, ?tmp$3: int)*, and foreign calls look like: *foreign llvm mul(tmp$2: int, 9: int, ?tmp$3: int)*, where the term *llvm* specifies the group which can possibly provide an instruction for the call. The compiler will decide what code to generate for a given *Prim*.

Conditional statements or goals partition a clause body as discussed above. But a loop conditional can't just do that as it would require a a jump back into a previous body. LPVM does not have these, and in fact these are part of the SSA drawbacks it wants to avoid. Loops are un-branched by generating new procedures which are involved in recursive calls. The calls are tail calls and hence the code generation will have to ensure tail call optimisation is enabled. A procedure will be generated for the loop body primitives which ends up calling itself recursively, and another generated procedure will contain the body which comes after the loop. The clause body that the loop was a statement in originally will end with a call to the former. There would also be a breaking conditional call to the latter.

If we consider *a*, *b*, *c*, *d* as a representation for one or more body statements or primitives, a looping construct looks like **do** *a b* **end** *c d*. The two generated procedures for it will can be **next1** : *a b next*1 **end** and **break1** : *c d* **end**. A more compound example with conditional breaking is shown in Figure 4.3. The guard goal(s) *b* decides whether to exit the loop or not, and *c* represents the remaining body of the loop after the condition. The conditional inside the loop split the generated procedure *next1* into *gen1* and *gen2*. As shown above, in the implementation the two clauses of *gen2* will be present in a *Fork* based on the value of *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 specifies 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. A function or procedure header should define all it's input and output types, along with any mutable resources that will be affected by it. 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 proto-

```
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 5.1: 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 5.2: Bool type as an Algebraic Data Type from the Wybe standard library

type. There is also no requirement for an interface or header file. 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.

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 *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 5.1. 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 5.2. 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.

Resources in Wybe are a way to specify some data that the procedure uses or modifies without explicitly being a part of the procedure arguments. This is an easy and pure way to list the side-effects of a procedure. An example of a resource in standard Wybe is *IO*. When a procedure says it is using the *IO* resource, we know it will be accessing some IO interface along with building its output parameters. For example: *public proc greet use io*.

Chapter 6

Transforming 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. Once the LPVM structure is obtained, we can run more optimisation passes over it. One such optimisation on a complete LPVM form is the *Expansion* pass. This pass explores costs and benefits of in-lining procedure bodies for calls, marking calls which should be in-lined. In-lining at the LPVM provides a lot of benefits which are apparent in the incremental features of the compiler also.

The type which stores the implementation of a Wybe procedure, in source and LPVM form, is shown in Figure 6.2. A procedure definition *ProcDef* will also contain other information about the callers, visible types, and more. A procedure can have multiple implementation, each implementation corresponding to a different *Clause* of the procedure. Initially a *ProcImpln* will be composed from the constructor *ProcDefSrc*, indicating source language form. On transformation to LPVM, the same type will have the constructor *ProcDefPrim*. The *Proto* and *Body* are similar to the constructors in Figure 4.2.

The compiler implementation keeps the pipeline modular. While the module implementations are stored in a *List* data structure, it is possible to generate a module dependency graph given a module. The implementation for any given module name can by pulled from or place in the module list store. The alternative approach of always maintaining a depen-

$$
\begin{array}{rcl}
func\ factorial(n : int) : int & \rightarrow & proc\ factorial(n : int, ?\$: int) \\
?c = bar(a, b) & \rightarrow & bar(a, b, ?c) \\
?y = f(g(x)) & \rightarrow & g(x, ?temp)\ f(temp, ?y) \\
proc\ greet(s : string)\ use\ !io & \rightarrow & proc\ greet(s : string, \#0 : io, ?\#1 : io)
\end{array}
$$

Figure 6.1: Normalisation of Wybe functions to Procedures.

$$\begin{array}{rcl} \textit{ProcDef} & \rightarrow & \textit{ProcImpln}* \\ \textit{ProcImpln} & \rightarrow & \textit{ProcDefSrc} \\ & | & \textit{ProcDefPrim} \\ \textit{ProcDefSrc} & \rightarrow & \textit{Stmt}* \\ \textit{ProcDefPrim} & \rightarrow & \textit{Proto Body} \end{array}$$

Figure 6.2: The Procedure Implementation Algebraic Data Type

dency *Map* would have made jumping into and out of module implementations more costly since these operations are done multiple times during a pass. This also makes loading and removing module implementations in the pipeline very easy, assisting the incremental features later.

Since LPVM procedure signatures should also present a pure interface like Wybe, Wybe *resources* are passed as a new parameter to the procedure. There is no explicit syntax since *resources* are a Wybe syntax feature. Hence, if a Wybe procedure uses the *io* resource, its corresponding LPVM clause will have an extra input and output parameter of the *io* type. This would mean that the LPVM procedure takes in an *io*, works on it, modifies it, and returns it. In a later pass, these common extra resource parameters could be optimised to point to just one common memory location.

6.0.1 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 6.1.

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.

6.0.2 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.

6.0.3 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.

6.0.4 Clause Generation Pass

By this pass the LPVM structure is more or less completed. The Wybe source semantics have been encapsulated in the LPVM IR, at least what is needed. The implementations for all the flattened clauses of the procedures will have the constructor *ProcDefPrim* applied, with the LPVM prototype and body primitives put in its place. Further optimisation passes will work on this structure until code generation.

6.0.5 Expansion Pass

An early optimisation that makes use of the LPVM structure is the in-lining of calls. This pass takes place on the clausal or LPVM form (*ProcDefPrim* constructor in the implementation). LPVM makes heavy use of procedure calls since every form of branching is essentially replaced with procedure calls. In-lining bodies of *callee* procedures removes most of these procedure call overheads. The decision to inline or not is based on a cost vs benefit analysis. This analysis takes into consideration the number of callers to that procedure, it's arguments, and body size.

In-lining benefits are only decidable if we have the LPVM clause form for that procedure. The structure of LPVM procedures is much more flat and limited as compared to Wybe source code, making copying procedure bodies easier, and the decision to do so more fine grained. For a procedure body to replace its call, the variables in the body have to be renamed while pasting. This renaming process is simpler with LPVM procedures since they list their input and output variables in the signature, and just these will have to be renamed. In-lining is one of the benefits of having LPVM forms of all modules involved in compilation in the compiler pipeline, and will be reflected in the incremental features later.

As an example, most of Wybe standard library types like *int ,bool, float*, have simple primitive procedures for binary operations. The implementation bodies for these procedures are just a single foreign instruction call to LLVM instructions. These will definitely be in-lined and in code generation phase will turn into a single LLVM instruction.

Chapter 7

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 (Feldman, 1979), 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 a future compilation processes in being incremental. It is this embedding that allows Wybemk to be an incremental and work-saving compiler.

By not building a target object file when the source is older, 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 beneficial 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 dependent 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 even though 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 incremental 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 dependent object files.

7.1 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. A special segment called *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 byte offset), which do not get mapped to memory. These are zero address segments. Using such such segment we can attempt storage of some useful serialised meta-data in the object file.

Our current implementation has the functionality to parse and embed information in *Mach-O* object files and *bitcode* files. The *Mach-O* file format is the Application Binary Interface (ABI) format that the OS X operating system uses for its object files. An ABI describes the byte ordering and their meaning for the operating system in this case. The embedded bytes do not interfere in any semantics of the object file. It still appears as an ordinary object file to every other machine utility or parser, like the tools *ld*, and *nm*. The only aspect which noticeably changes is it's total byte size.

7.1.1 Mach-O Object File Format

Quite simply, an object file is a long byte string sequence. Referring to the Apple documentation[1]on their format, we are able to parse and edit the *Mach-O* byte structure. The first 32 bits or 4 bytes are considered to be the magic number if read in little endian format. The magic number constant determines what kind of ABI the rest of the bytes of the file follow and their ordering. On OS X we can have 32 bit and 64 bit *Mach-O* object files and Universal binaries. Universal binaries or Fat archives contain more than one object file. Wybemk is interested in *Mach-O* object files.

The header bytes of the identified structure will give the number of load commands the file contains. The load commands are the segments which get loaded to the main memory during execution. The load commands provide the name of their segment and a pointer to offset of their data. Each segment can also contain multiple sections. Wybemk creates a new segment called *__LPVM* and a section in it called *__lpvm*, following naming conventions. Once the offsets are correctly determined and bytes describing the load command put in its right place, we can insert any length byte-string at our offset. This byte-string will be the encoded serialised form of our embedded data.

While we have only covered the *Mach-O* object files in our embedding implementation, it is also possible to use the system *ld* linking, found on most *UNIX* OS, to add new segments and sections. Other object file formats, like the *Elf* format for Linux, can be used for embedding this way. This is part of our future planned work.

7.1.2 Bitcode Wrapper

LLVM IR can be put in *bitcode* files[2]. These are binary representations of the LLVM IR it encodes. These files are identified by a magic number, similar to a *Mach-O* file. LLVM tools can easily generate these formats and even compile them to object file and link them. In a way they are a machine architecture independent version of object files and can be easily distributed between compilers which support LLVM. It provides great interoperability. Hence we want to have the option to utilise them in a similar fashion to how we utilise object files.

A *bitcode* wrapper differs from an ordinary *bitcode* file in its initial magic number. This wrapper format specifies a header which should give the byte offset of the byte string representation of the LLVM IR it holds. The rest of the bytes after the header and before the offset are therefore free to embed information. We use this space to store our LPVM information

[1]Documentation `https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/MachORuntime/`

which usually goes into the object file.

7.2 Incremental Compilation

Wybe, as a programming language, wants to be useful for large scale projects. Thus, it wants its compilation process to be as effecient as possible. In larger projects, a large number of modules are involved in a single build. Doing incremental builds would involve smaller changes being added for every fresh build. Having all the modules compile again is a waste of time.

The goal is to make Wybemk incremental at multiple levels. This can be done by identifying key stages of a compilation process which can act as save and restore points. The saving is done in object files (or bitcode files) as shown above. The decision to restore has to be a careful one, as a false positive would result in a completely wrong build. There should be no margin for these kinds of errors to exist if this compiler is to be used in production builds. There is also a requirement of being incremental without losing the benefits of LPVM optimisations. With these constraints, currently Wybemk has two incremental and work saving approaches: Module level reloading, and storing hashes of key compilation stages.

7.2.1 Module level reloading

Wybemk compiler behaves like GNU Make but it does not depend on Makefiles. It infers the module dependancy graph and builds everything accordingly. It should also link in standard libraries and external foreign libraries wherever needed. Inferring the dependancy graph is done through parsing the top level *use module* statements. During compilation LPVM representation of each module in the graph is built. For modules who have a newer object file, this representation will be stored in a serialised form and has to be *reloaded* into the pipeline. This keeps the optimisations going and allows other modules to inline functions or procedures from that module.

For example the *int* module in the Wybe standard library module (Figure 5.1) mostly has one line procedures and functions (simple arithmetic operators pointing to LLVM instructions). That is, their body is a single procedure call. Instances of calls to *proc* + can be replaced with the body proc call instead. And this is what actually happens when the standard library object file is *loaded* by the Wybemk compiler. Inlining at LPVM level provides these small but essential benefits.

We want to embed only the minimum set of data we will need for the next compilation. The serialised abstract data type in the implementation is the subset of the type which holds information on the whole *Module*. The complete *Module* type holds information on the exported types, procedure

[2]See `http://llvm.org/docs/BitCodeFormat.html`

implementations, sub-module names, dependency information, LLVM implementation, among others. The serialised subset will remove local information and just preserve the interface of the *Module*. For example, the LLVM implementation is already present in essence in the *TEXT* segment of the object file in object code form.

Not every procedure defined in the module has to have its LPVM form passed along in the serialised *Module*. Private procedures cannot be inlined by any other module and hence will never be utilised in an intermodule optimisation. Serialising the LPVM form instead of the source code form already saves a lot of space, having extra space saving heuristics is a bonus. The exportable *Module* subset is serialised as a byte string. A LPVM primitive is made up of either of two constructors (Figure 4.2): *PrimCall* or *PrimForeign*. This keeps the tag byte size small for serialising a procedure body. Even the procedure LPVM implementation is made up of only one constructor: *ProcDefPrim*. The tag byte contains flag type information to identify which constructor to use while decoding.

7.2.2 Incrementality through Hashing

Certain stages in the compilation pipeline can act as checkpoints where Wybemk checks if it is going to do the same work as the last compilation. These checkpoints have to be chosen carefully as having numerous checkpoints will start slowing down the compiler instead of saving time. Certain stages behave as natural checkpoints, like the end of the parsing and the end of transformation to LPVM. For determining if Wybemk has reached the same stage as before, it hashes the serialised form of the current representation at hand, and compares it with the hash stored in the object file. Hence, along with the serialised LPVM representation of module, Wybemk object file also embeds certain hash strings in a serialised map data structure.

The constraints under which our system is to operate is similar to the ones mentioned in Cooper and Wise (1997) for their C++ incremental compiler system. We choose our granularity in such a way that we can automatically generate dependencies for it. We have to absolutely sure we are not executing old code and that we are choosing the right old versions. Their system uses time stamping as a heuristic to choose between compiling or not, while we are using hash comparisons to choose between compiling or loading.

These methods are meant to kick in for scenarios where the source file has more edits (or is newer) than the object file. In these cases simply loading the whole LPVM module is not correct. But identifying unchanged parts and loading those parts while re-doing the other parts provide a faster compilation time. In our current iteration of the compiler we are applying hash comparisons at two stages: after parsing, and just before the LPVM optimisation passes.

After parsing we can store the hash of the parse tree generated by the parser. When the parser finishes its work, Wybemk can hash its AST and compare it with the hash of the previous AST. For source code edits involving changing or adding comments and white-space, the parse tree doesn't change even though the source file will now have a newer modification time. These trivial yet extremely common edits should not run the whole compilation process. In this case we just load the final LPVM form from the object file. A future extension to this check is considering procedure, functions, and other top level items without order, so that that they all are individually checked and changing their order in the file changes nothing.

Completely change this last part.

The choice of granularity for the Wybemk compiler is a Wybe procedure or function. As shown in chapter 6, a Wybe procedure or function is flattened to a procedure form very early in the compilation pipeline. The dependency resolution is then narrowed to tracking the relationship of these flattened procedures to the LPVM procedures they are transformed to. A Wybe *procedure A* depends on *procedure B*, if it contains a call to *B*. Flattened Wybe procedure can be transformed to one or more LPVM procedure.

If one procedure is changed, the only parts that should be recompiled is that procedure itself and some other affected procedures (due to inlining or calls). We can check for edits at the parse tree level (in Wybe form) or when the LPVM form is generated (before optimisation passes). Checking the Wybe source form coincides with the parse tree hashing we are doing above. Every Wybe function or procedure is flattened to be a procedure, so when we talk about procedures, we are considering all top level items in the module.

If we change a variable name uniformly throughout a procedure body, it's LPVM form would not change since LPVM will be generating it's own unique variable names. These kind of edits don't change the semantics of the procedure and re-compilation is not needed. Hence, it is useful to check LPVM forms of a Wybe procedure and on a successful match, we can skip the further optimisation passes, loading the final form and object code for those procedures. A Wybe procedure, when transformed to LPVM form, might create more than one LPVM predicate or procedure. This happens for conditional branching and loops. This mapping has to be tracked so that we know which LPVM procedures to load in.

Chapter 8

Code Generation to LLVM

We do not generate machine code directly using LPVM IR. Instead we are hooking into the LLVM project so that we are able to compile Wybe programs on multiple architectures without additional effort. Without the LLVM we would have to write code generators for every popular architecture and more. LLVM is a popular open source project. A lot of older compilers (and new) are being re-targeted to LLVM IR, and the efforts taken previously in making the machine code generator efficient is being poured collectively into LLVM. We feel it's worthwhile to do the same for Wybe.

The LLVM IR has an abstract structure of an imperative program, and is a SSA based form. Earlier we talked about how LPVM IR solves the drawbacks of the using SSA naming scheme and virtual functions, and yet we are ultimately targeting a SSA based IR. We do this based on the observations that we are doing majority of our optimisation and program analysis in the LPVM stage, and the simplified LPVM form does not need the *φ-function* to be present in the LLVM IR. In its final stage, LPVM procedure body primitives behave like calls to other procedures or virtual instruction calls. Generating LLVM IR on these bodies is very direct. We take a deeper look at these conversions in section 8.1.

Using the compiler tools provided by LLVM framework, we can create object code from the IR easily. For accessing instructions not provided by LLVM, we link in a shared C library. Since the C compiler *clang* compiles C using LLVM, we can either join the LLVM form of this library with ours or just using the system linker to link object files. Using this library we can provide a stronger support for compound types, as discussed in section 8.2.

8.1 Transforming LPVM to LLVM

Since LLVM is based on an imperative form, we have to transform each LPVM procedure back into an imperative function. Wybe and LPVM IR support procedures with multiple outputs, but imperative functions do not do so by default. This flexibility should be reflected in the LLVM IR as well. Having multiple outputs are an abstraction of placing values in multiple registers at the end of the execution of a procedure body. In LLVM each register is virtually reflected by a variable name or an aggregate structure. We wrap the multiple values which are to be returned by a procedure in a register structure. The values are unwrapped in the body of the caller in the same order.

Referring to the structure in Figure 4.2, we know that a LPVM primitive statement is either a LPVM *local* procedure call or a *foreign* call. We said these calls are meant as a directive for the compiler to generate specific machine code for. For a *foreign* call of the form: *foreign group proc_name(..args..)*, we expect an *instruction* provided by the given *group* to replace the call in the generated LLVM IR. Currently in our implementation, we have three types of foreign groups: *c*, *llvm*, and *lpvm*. Foreign calls with the group *c* refer to functions defined in the shared C library file which is linked in later. In the LLVM IR these are called by declaring this function name as an *extern* and making a *call* to it. Foreign calls with the group *llvm* refer to LLVM instructions directly by name. The *args* will be type tested and the validity of the call ensured at code generation again. Foreign calls to *lpvm* are special functions whose underlying implementation can be provided by anyone. Currently we have memory allocation and mutation functions in this group, and we use *C* functions to provide the object code.

The *GCD* function that we showed the LPVM transformation of earlier, is shown being translated to LLVM in Figure 8.1. The two clause branches spawned off the condition on *b* end either in a *return* instruction and a procedure call respectively. An *foreign* call like *foreign llvm add(a:int, b:int, ?c:int)* will have to converted to a function call. This call translates to the LLVM IR instruction *add* which adds two integers. Transforming a procedure call to a function call is quite trivial in most scenarios. The function call version for the above example is: *c = add i32 a, b* (actual variable names look much different in LPVM and LLVM).

LPVM replaces loops with tail call recursion. For LLVM IR to generate efficient machine code, it is extremely important to ensure that the LLVM tail call optimisation is turned on. This requires turning on certain flags in the *call* instruction generated in LLVM IR, and ensuring the call is indeed a tail call in the imperative function body representation of the procedure.

8.2 Code generation for Wybe types with LLVM

LLVM provides access to the basic integer type of arbitrary bit precision. We are considering an *int* to be of the machine word size, which is more commonly *i32* or *i64*. LLVM also has floating point types. Using these we can model all of Wybe's basic types as shown in chapter 5. Compound

```
$\mathbf{gcd}(a, b, ?ret) \rightarrow$
            $foreign$ **llvm** icmp ne$(b, 0, ?tmp1)$
            $case$ tmp1 $of$
        0 : $foreign$ **llvm** move$(0, ?ret)$
        1 : $foreign$ **llvm** urem$(a, b, ?b')$
            $gcd(a, b', ?ret)$
```

LLVM

```
define i64 @gcd(i64 %a, i64 %b) {
  %0 = icmp ne i1 %b, 0
  br i1 %0, label %if.then, label %if.else
if.then:
  %1 = urem i64 %a, %b
  %2 = tail call i64 @gcd(i64 %a, i64 %1)
  ret %2
if.else:
  ret i64 0
}
```

Figure 8.1: Transformation of the LPVM form of *GCD* to LLVM

types and Abstract data types require an interface to the heap. An elementary way to do this is use LLVM function calls to make calls to C's *stdlib*. Namely calls to *malloc* and *free*. LLVM's instruction *getelementptr* can index allocated memory in the heap. But we also want to make headway into providing garbage collection in the compiler. Without providing a complete solution, we can still access dynamic garbage collection by making use of the *Boehm GC*. Accessing the C library version of this through the shared library we link in later, we can use the replacement function for *malloc*, called *gc_malloc* to enable the Boehm garbage collector to automatically collect memory. To use these calls, LPVM generates *foreign* calls in the *lpvm* group: *foreign lpvm alloc(8:int, x:vector)*, where *vector* is an example new type name.

For algebraic data types, the compiler will determine the byte sizes (or word sizes) for every type constructor used and will generate access and mutate calls accordingly. Both classes of procedure calls translate to a combination of LLVM *getelementptr*, *store*, *access* functions.

Bibliography

Alpern, B., Wegman, M. N., and Zadeck, F. K. (1988). Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 1–11, New York, NY, USA. ACM.

Appel, A. W. (1998). Ssa is functional programming. *SIGPLAN Not.*, 33(4):17–20.

Cooper, T. and Wise, M. (1997). Achieving incremental compilation through fine-grained builds. *Software: Practice and Experience*, 27:497 – 517.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490.

Feldman, S. I. (1979). Make. *Software: Practice & Experience*, 9:255 – 265.

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.

Johnson, M. (2008). Note on intermediate representation. `http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/16-Intermediate-Rep.pdf`. Lecture Handout on the DragonBook.

Lattner, C. (2002). LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL. *See* `http://llvm.cs.uiuc.edu`.

Ottenstein, K. J., Ballance, R. A., and MacCabe, A. B. (1990). The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. *SIGPLAN Not.*, 25(6):257–271.