An Incremental and Work-Saving Compiler for the Wybe Programming Langauge

Ashutosh Rishi Ranjan (709065)
Supervisor:
Dr. Peter Schachte

Research Project: 75pts
MSc. Computer Science: COMP60002

6th June, 2016

Contents

1 Abstract **6**

2 Introduction **7**

3 Literature Review **10**
3.1 LPVM: Horn Clauses as an Intermediate Representation . . 11
3.2 Other smart recompilation strategies 15
3.2.1 Decreasing compilation granularity 15
3.2.2 Smart Recompilation 16

4 Wybe Programming Language **19**

5 Implementation of LPVM **23**

6 Transforming Wybe to LPVM **26**
6.0.1 Flattening Pass 27
6.0.2 Type Checking Pass 29
6.0.3 Un-branching Pass 29
6.0.4 Clause Generation Pass 29
6.0.5 Expansion Pass 30

7 Wybemk, Compiler and Build System **31**
7.1 Storing structures in Object files 32
7.1.1 Mach-O Object File Format 33
7.1.2 Bitcode Wrapper 34
7.2 Incremental Compilation 34
7.2.1 Module level reloading 35
7.2.2 Incrementality through Storing Hashes of Stages . . . 36
7.3 Examples of Gains 38

8 Code Generation to LLVM **40**
8.1 Transforming LPVM to LLVM 41
8.2 Code generation for Wybe types with LLVM 44
8.3 Linking Dependencies 45

9 Conclusion and Future Work 47

List of Figures

3.1 A Simple C-like *GCD* function 14
3.2 Comparison of the SSA and LPVM clause forms for the GCD function 14

4.1 Sample of the *wybe.int* module from Wybe standard library . 19
4.2 Bool type as an Algebraic Data Type from the Wybe standard library 20
4.3 Sample Wybe programs 21

5.1 Forking in the *Clause* on a *Guard* conditional 24
5.2 LPVM Implementation Data Type 24
5.3 LPVM Procedures generated for an imperative loop 24
5.4 Comparing the clause form of GCD (left) with its actual LPVM implementation (right) 25

6.1 Normalisation of Wybe semantic structures to Procedures . 26
6.2 The Procedure Implementation Algebraic Data Type 27
6.3 Normalisation: generation of procedures to provide an interface to a user type 28

8.1 Comparison of the LPVM form of *GCD* to its LLVM form . . 43
8.2 LPVM Procedures generated for an Abstract Data Type . . . 45

Foreword

I want to thank Dr. Peter Schachte for giving me the opportunity to work with him on his new programming language project, the Wybe language and compiler. It is a great learning experience and immense fun. I wanted to use this section to highlight my exact contributions to this compiler code base, since it was already well under construction before I started my research project, and the discussion in this thesis covers more than just my contributions.

I joined the compiler project at the stage when the entire front-end and LPVM Intermediate Representation transformation was already in place. Since then, I have worked on extending the compiler code base to include translation of the logic IR, LPVM, to the LLVM IR and the machine code generation from that. With the guidance of Dr. Schachte, I have also been working on adding the incremental features, discussed in this thesis, into the compiler, including the work on the build system. Essentially our work flow is divided at the LPVM IR stage. Dr. Schachte works on extending the language and its translation to LPVM, whereas I work on the rest of the back-end stages thereafter, incorporating his changes and extending the incremental features.

Chapter 1

Abstract

Compiler efficiency in an optimising compiler can be increased in a variety of ways. The usual optimisation methods target the Intermediate Representation (IR) and the code generation phases to achieve better memory and time performance. The compilation time for a build is also dependent on the number of passes the compiler will make over its internal representations, and the number of source modules it has to compile. This thesis explores different ways to reduce build times by making the build process more incremental and work-saving so that a compiler can obtain a net gain in compilation turn-around-time by avoiding redundant work over a sequence of builds. These features are a part of the compiler and build system *Wybemk*, for a new programming language called *Wybe*. A significant portion of the incremental features is derived from the use of a new logic language based Intermediate Representation, called *LPVM*, in the compilation pipeline. Modelling the build system after the GNU Make utility, the *Wybemk* compiler utilises the structure of an object file to embed meta data into it, which is used in subsequent compilations to reload redundant work. *Wybemk* includes an *LLVM* back-end to provide code generation.

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 source language for this compiler is a new multi-paradigm language called **Wybe** (pronounced *Wee-buh*). The compiler eventually targets the **LLVM** framework which allows it to be flexible in targeting multiple modern architectures and actually generating machine code. Ideas explored in this thesis stem from the motivation of achieving efficiency in a compilation process by avoiding any work that has been done in an earlier compilation. The term *work-saving* here, means that the compiler should be able to recognise scenarios where it can reload completed previous work rather than do the same steps again. For any particular build it should recompile only those source files which really need to be recompiled. Being *incremental* means being able to define a complete compilation as a function of smaller individual compilations, and then working on only those smaller units which have changed. The units which have not changed need not be recompiled. The level of this smaller compilation unit is the *granularity* of a compiler. Traditionally, a compiler operates at the granularity of a source file. We want to work with something smaller than that, without interrupting normal operations of a compiler.

The purpose of the Wybe programming language is to be an easy to learn language without sacrificing robust static type systems which is often sought after in a real world project. Wybe wants to serve the same domain as languages like Python, in being easy to read and reason with, and statically typed languages like Java, in being suitable for production systems. It also brings with it the power of declarative languages and efficient compiled code. A work-saving compiler for Wybe, which is geared to eliminate redundancy, should further push its suitability for projects involving a large number of modules. With LLVM as the back-end code generator we can produce more efficient machine code as the project moves ahead to maturity.

The choice of an intermediate representation for use in the compiler is an important one. It is the data structure which holds an abstract representation of the source code semantics, while being machine independent. It undergoes multiple optimisation passes which may or may not alter its structure, and will ultimately be targeted to generate code for any given architecture. The choice of the IR structure and form makes certain program analyses easier to do than the rest. A particular form may even enable some specific transformation technique which other forms might not inherently allow. In a compilation pipeline, there can multiple IR forms between the source code and the machine code (Johnson, 2008). IRs which resemble the source code semantics more closely are deemed to be on a higher level than the ones which have a closer resemblance (and the restrictions) to some machine code or assembly. An IR will strip away information it does not need anymore, or information that it has utilised completely, and transform to a flatter, lower level form as it moves through a compilation process. In our pipeline we utilise two different forms of IR. The **LPVM IR** form is generated from a 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 passes) and is mostly needed so that we can generate machine code without duplicating the effort in the numerous code generators the LLVM project already includes.

LPVM IR is the implementation of the logic programming based IR form given in Gange et al. (2015). The term *LPVM* is an abbreviation for *Logic Program Virtual Machine*, and named in a similar style to *LLVM*. They are not really virtual machines, but as an IR, they provide an abstract instruction set resembling an abstract machine. While the use of LPVM inherently provides several benefits in program analysis and optimisation, we can also exploit its structure to achieve the incremental features in our compiler. LPVM has an incredibly simple IR structure which makes doing program analysis and reasoning with the LPVM form easier than its counterparts. It also allows the use of powerful logic programming analyses. In this thesis though, the focus is on exploring the structural benefits of LPVM in providing incremental features in the compiler, rather than its proposed logic programming optimisation techniques.

The Wybe compiler is called **Wybemk** (pronounced *Wee-buh-mik*). It is a combination of the words *Wybe* and *Make*. To eliminate redundant recompilation, we need some internal compiler structures passed on from a previous compilation to be compared with the current compilation. By identifying parts which can be re-used in the current compilation process we can skip a lot of passes. To facilitate this forwarding of information we make use of the byte structure of an object file. Since building relocatable object files is the natural end goal of any compiler, we don't have to develop new Wybemk specific container formats. Hence, we explore ways of embedding information into architecture dependent object files. These object files will appear as ordinary object files to every other compiler, but will contain extra docile information usable by Wybemk.

Another focus of the Wybemk tool is to have a build system ingrained into the compiler. We try to mimic the GNU *Make* utility (Feldman, 1979) with some simplifications. To behave like Make, we recognise object files which are newer than the corresponding source file and avoid re-building that object file. But we also don't want to depend on external Makefiles and interface/header files. Wybemk is intelligent enough to determine the transitive closure of dependencies to build for any give source or target. The goal here is to provide a complete solution for a build system so that there is no need to rely upon third party tools and build systems later when the language matures.

The rest of the thesis presents a more in-depth discussion of the points made above, and the work done to build this 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 discussions on the important papers which form a base for this thesis. Especially the paper introducing LPVM. The Wybe programming language is then introduced in Chapter 4. Having an overview of the actual language being compiled makes discussion easier. In Chapter 5 we present the actual implementation of LPVM used in the compiler, and in Chapter 6 we discuss the transformation of Wybe to LPVM. This is the first stage of the compiler. We are then set to present the incremental and work saving build system, Wybemk, in Chapter 7. Finally we discuss code generation from LPVM to LLVM in Chapter 8. Chapter 9 presents the future direction of Wybemk and concludes.

The thesis is tangential to a working implementation of the Wybemk compiler, which is written entirely in Haskell.

Chapter 3

Literature Review

The logic language based intermediate representation presented in Gange et al. (2015) is the IR used in the Wybe compiler. It is an integral part of the compilation pipeline and its features and structure 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. An in-depth discussion of this paper is presented in Section 3.1.

Exploring ways to be smarter about recompilation is a common way to add to the efficiency of a compiler, independent of the optimisation gains in the actual compilation process. A successful compilation process on a statically typed language ensures static type safety. Circumventing this process in a bid to eliminate redundancy should be done carefully. The advantages of avoiding redundant compilation in a lifetime of a project is very noticeable (Adams et al., 1994). There are a lot of different approaches to this, and we derive ours by observing these. Discussions on some smart (re)compilation processes is presented 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 building process runs 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. By reading the file modification time we can derive an elementary heuristic to avoid redundant re-building of a target.

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 back-end. Many popular languages like Haskell, Rust, etc. are developing LLVM back-ends 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. The convenience of transforming an IR to another abstract high level IR which has code generators already written for it, is incontestable.

A recent example of incorporating a simpler IR form in the middle of the compilation pipeline, before LLVM, is the Rust programming language. Rust is a new, type robust, systems programming language which is gaining a lot of steam. In their pipeline they have introduced a new *middle-level* IR called *MIR* (Matsakis, 2016). In their motivation to add another IR before the LLVM IR they make an observation on the utility of a simple IR structure in incremental compilation. They are using MIR to factor out redundant work by doing a similar saving and reloading of IR to disk, like Wybemk. Apart from Rust compiler, the compiler for Swift Programming Language is also incorporating its own form of IR before the LLVM IR stage. This is called the Swift Intermediate Language (SIL) (Groff and Lattner, 2015). Swift programming language is the new language Apple is using for building applications for its platforms. The author of this language, Chris Lattner, is the same author behind the LLVM project. A lot of the work done on the Swift programming language is also contributed to the LLVM project. Yet, Swift source is first transformed to SIL before it is finally transformed to LLVM.

3.1 LPVM: Horn Clauses as an Intermediate Representation

Gange et al. (2015) presents 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 structure in building incremental features in the compiler, it's important to understand the reasoning behind the structure of LPVM and the benefits it has 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 of these drawbacks from the get-go. This structure uses *Horn Clauses* from logic programming, imposing certain limitations on a general logic language.

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 with 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 *virtual* and will not have code generated for it. Instead it's evaluation is mainly for program analysis and requires backtracking into the 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.

To determine the control flow path to a *φ-function* without looking into a predecessor block, an extension to the SSA form, called the Gated Single-Assignment (GSA) (Ottenstein et al., 1990), was created. This extension 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 as part of block signatures.

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) given in Ananian (2001). This form includes another virtual function (σ) at the end of the blocks which branch into alternate blocks, describing the destinations of each alternate variable. Again, this information is easily determined in the LPVM form.

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 presented, 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 presents the in-flowing and out-flowing parameters for its block equivalent, making local analysis much simpler.

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 the 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 every 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. We will see it's implemented structure in Chapter 5.

At the top level of an LPVM form there are only predicates (or procedures). In the form presented in the paper a procedure consists of multiple *Horn Clauses*. The *Head* of a *Clause* describes its parameters and the predicate/procedure name it belongs to. These parameters can be in-flowing or out-flowing. In the abstract model the output parameters are separated from the input parameters with a semi-colon. 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. There may be two alternative *Clauses* with the same name having switched up modes for the same parameters. Since determinism will select only one of them as the actual 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 chapter 5, 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 3.2 demonstrates these differences of semantic structure between the SSA (on the left) and LPVM *clause* form (on the right) for a sim-

```
int gcd ( int a, int b )
{
  while ( b != 0 ) {
     int temp = b;
     b = a % temp;
     a = temp;
  }
  return a;
}
```

Figure 3.1: A Simple C-like *GCD* function

entry :
br header

tail :
return a_1

header :
$b_1 = phi(b, b_0)$
$a_1 = phi(a, a_0)$
$if\ (b_1 \neq 0)$ body tail

body :
$t_0 = b_1$
$b_0 = a_1\ mod\ t_0$
$a_0 = t_0$
br header

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

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

Figure 3.2: Comparison of the SSA and LPVM clause forms for the GCD function

ple *gcd* function (Figure 3.1). The *gcd* function takes two numbers a and b (where $a < b$) and computes the *Greatest Common Divisor* between them. We will follow this example until its final conversion to LLVM in this thesis. The *header* block in the SSA section is the loop header. This block is entered at the start of the program (*entry*) or on the next iteration of the loop (from *body*). Both these converging branches will bring their own version of b and a, and hence a *φ-function* is needed. The *clause* form on the right looks much simpler. It presents two *clauses* of *gcd* predicate (or procedure). The first one is called for $b \neq 0$, and the other for its complimentary evaluation. Parameter passing deals with different versions and values on each iteration (recursion here). We don't need different names for the same variable.

3.2 Other smart recompilation strategies

Our goal is to build an incremental compiler which exploits the structure of the LPVM IR at compile time. To achieve this we want the granularity of the compilation to be smaller than that of traditional source files. If we can isolate and identify object code for certain blocks in the source file, we can define the compilation process to be composed entirely of these smaller blocks. An incremental compiler for C++ given in Cooper and Wise (1997) presents a working system which compiles at the function (or object) level as opposed to the file (or module) level. Their approach is discussed in section 3.2.1.

Another approach for avoiding unnecessary recompilation is analysing the change in context of a module that an source edit causes and accordingly compiling only modules affect by that change (Tichy, 1986). A context is defined as an interface of a module which can be manually or automatically generated in a compilation process. This approach is discussed in section 3.2.2.

3.2.1 Decreasing compilation granularity

The system described in Cooper and Wise (1997) is an incremental interactive program development system for C++ called *Barbados*. It features a compilation system with a granularity focused at the level of functions and other top level structures instead of the usual file level granularity in other C++ compilers. This is quite similar to the build system we want for Wybe, without the interactive part. The requirements for building such a system, as listed in the paper, involve automatic dependency inference for the chosen granularity, transparent compilation, and ensuring no old code is ever executed. These are also the constraints we want the Wybe compiler to work with. The functioning 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 without exploiting intermediate structures, 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 necessary 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 time stamps are part of the interface for all basic entities. Although using time stamps to determine the need for recompilation is simple and effective.

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. Every compilation is done in a declarative fashion over this tree. Starting from the root, the system moves through the tree until it reaches a leaf and then works it way back from there. This way it can ensure that it is dealing with dependencies in the correct order. There are a couple of problems that can arise with this traversal, such as circular dependencies and the change of dependencies during tree traversal. These problems are solved in Barbados by doing multiple traversals of the tree until a stable state is reached. The paper finds that the time spent in multiple traversals is negligible when compared to compilation times, providing support for this approach. However, it does feel wasteful. 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. We can observe that these cases are few, but since they are so trivial it should be checked. In our system we also use hash comparisons to detect changes instead of time stamps. It is feasible for Wybemk as the LPVM structure is very limited and easy to hash.

3.2.2 Smart Recompilation

The goal and motivation of this implementation is similar to Wybemk: avoid redundant recompilation on minor changes. Similar to the C++ incremental compiler (Cooper and Wise, 1997), being able to maintain a complete dependency relation reliably is important. This relation is maintained between contexts of compilation units involved, and on determination of a change, only the dependent or affected units are recompiled. Contexts here are similar to C header files which expose an interface of a module. A module here refers to the smallest block the source language compiler can compile independently and for which a context can be generated. Thus if another module *includes* a context, it is dependent on that context. Unlike our system, the determination of a change here is done by a version control system. A module context is said to have changed if the controller determines a revision on the old context. This can be more effective at pointing out editing changes which didn't actually change anything except the last modification time. In our system, we try to deal with these scenarios with internal hash comparisons instead of being reliant on an external version control system.

Although, the behaviour of this external controller is shown to be very useful. It is a combination of the Make utility and a Revision Control System. Apart from tracking the compiled object files, it maintains a list of attributes for them. These attributes contain history of compilation and past compilation configurations. When a target object code is to be compiled, it behaves in a Make like fashion by recursively visiting the object files in the target's past configuration attribute to determine the changed contexts. It updates these attributes on a re-compilation or a re-use accordingly. When multiple contexts have changed, on which the target depended on, each context is visited one after the other and checked in a similar fashion as the target.

While the ideas explored in the above system skip a lot of redundant recompilation steps, it is limited by the granularity of the source language. Even the paper notes this by realising that it won't be able to eliminate all redundant compilation on smaller changes. Also, on multiple context changes, the system could have adopted a better traversal methodology. The current traversal is not accommodating inconsistencies arising in the middle of traversal. In contrast, the system described in (Cooper and Wise, 1997) does multiple traversals to deal with any new information generated in the middle of the dependency traversal. But overall, the smart compilation algorithm is able to ensure that the correct compiled code is being used.

Improving on this smart recompilation algorithm, a smarter recompilation approach is presented in (Schwanke and Kaiser, 1988), which relaxes the consistency criteria so that more re-compilations can be avoided. However, in their approach a prompt to the programmer is required who selects a subset of the source files affected by a change to continue with the algorithm. This affects the transparency requirement we want, and which Cooper and Wise (1997) insists upon as well.

The above strategies have shown a common focus on generating accurate dependencies for a chosen compilation unit. Avoiding recompilation of a complete module on no change is also a common and easy to implement strategy. While the base ideas incorporated in Wybemk are similar to the above strategies, we have a new and different intermediate structure to play around with. Our approach will naturally be a little different.

Chapter 4

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 its 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, similar to a function in other declarative languages. 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. The procedure header lists all the important 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**. Both functions and procedures are able to present a pure interface. A function or procedure header should define all of it's input and output types, along with any mutable resources that will be affected by it. By also forcing

```
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 4.1: 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 4.2: Bool type as an Algebraic Data Type from the Wybe standard library

the information flow to be explicit, Wybe makes it easy to determine the effect of a function or procedure just by looking at its prototype. 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). Thus, a procedure header which looks like: *proc* $mul(a, ?b, c)$, says that the procedure *mul* operates on two input parameters at positions *a*, *c*, and builds one output at parameter position *b*. There is also no requirement for an interface or header file. With procedures having a preceding visibility label (*private* or *public*), it is very easy to automatically generate an interface for a module.

A Wybe **module** is equivalent to a Wybe source file with the extension *.wybe*. The module name is the same as the source file name, but without the extension. A module interface consists of the public functions and procedures defined in it. There is also a separate syntax to declare sub-modules inside a module. Sub-module names are qualified with the outer/parent 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 entire source file *A.wybe*. This also means that importing module *A* will give you access to its sub-modules. Defining new types also creates a new sub-module named after that type. All dependencies of a module can be inferred through the top level *use module* statements in the source file. Sub-modules have the dependency of the super-module automatically added to their interfaces. Hence, a sub-module can access the interface of its sibling modules. When *module A uses module B*, *module A* can call all the public functions and procedures of *module B*.

A Wybe module can contain statements outside any procedure, as top level statements. These are always executed in order when the module is called or imported. The idea here is to insert module initialisation and setup code here which is required to run even on imports. For example, a module defining an interface to access a database can have automatic initialisation code in top level statements to set up the connection to the database.

Simple Statement

```
println("Hello, World!")
```

Factorial Function

```
public func factorial(n:int):int =
    if n <= 0 then 1 else n * factorial(n-1)
```

Factorial Procedure with IO

```
public proc factorial(n:int) use !io
    ?result = 1
    do while n > 1
       ?result = result * n
       ?n = n - 1
    end
    println(result)
end
```

User Types

```
public type position = public position(x:int, y:int) end
```

Figure 4.3: Sample Wybe programs

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 just requires 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 4.1. 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* too. The *bool* type can be defined as an ADT with two type constructors, as shown in Figure 4.2. 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 overloaded 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. The above definitions model multiple parameter modes of a logic language in a way. Having the three definitions as above, the procedure *add* can be called with any single of it's parameters un-bound (as an output).

Resources in Wybe are a used 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*. Custom resources can also be defined.

On the imperative side, Wybe does have loops. Figure 4.3 shows a few examples of Wybe programs to introduce the general syntax. The procedure form of the factorial operation reveals that it will be doing IO in its body using the *use !io* syntax.

Chapter 5

Implementation of LPVM

The actual implementation of the LPVM structure differs a bit from the abstract structure proposed in Gange et al. (2015). When a guard goal is encountered in a clause body, the abstract representation rules result in another clause being created with the same sequence of goals up until that guard goal. In the implementation however, to avoid duplication, there is only one clause generated. Assuming that a basic clause body is the sequence of goals barring a guard goal, then at the guard goal a fork is created which contains the fork condition, and a list of subsequent basic clause bodies. Each of the basic clause body in the list corresponds to a different evaluation of the fork condition (Figure 5). Thus, 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 into another block and hence there will be no need for *phi* like functions.

The Figure 5.2 shows the algebraic data type used to hold the LPVM IR in the compiler implementation. 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 is 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*.

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* created for a different combination of modes of the procedure parameters. For example, the procedure $-$ can have clauses $-(a, b, ?c)$ and

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

Figure 5.1: Forking in the *Clause* on a *Guard* conditional

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

Figure 5.2: LPVM Implementation Data Type

$-(a, ?b, c)$, 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 as such this construct directly enables it it.

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 jump back into a previous body at the end of an iteration. 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 that the tail call optimisation is enabled. There are two new procedures generated in the un-branching: The first will contain the loop

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

$$\begin{array}{rcl}
\textbf{proc}\ next1 & \rightarrow & a\ gen1\ \textbf{end} \\
\textbf{proc}\ gen1 & \rightarrow & \textbf{guard}\ b\ 1 : break1\ \textbf{end} \\
 & \mid & \textbf{guard}\ b\ 0 : gen2\ \textbf{end} \\
\textbf{proc}\ gen2 & \rightarrow & c\ next1\ \textbf{end} \\
\textbf{proc}\ break1 & \rightarrow & d\ \textbf{end}
\end{array}$$

Figure 5.3: LPVM Procedures generated for an imperative loop

$$\begin{aligned} gcd(a,b,?ret) \rightarrow\ & \textbf{guard}\ b \neq 0 \\ & \wedge mod(a,b,?b') \\ & \wedge gcd(b,b',?ret) \\[1em] gcd(a,b,?ret) \rightarrow\ & \textbf{guard}\ b == 0 \\ & \wedge ret = a \end{aligned}$$

$$\begin{aligned} & \textbf{gcd}(a,b,?ret) \rightarrow \\ & \quad foreign\ \textbf{llvm}\ \text{icmp ne}(b,0,?tmp1) \\ & \quad case\ \text{tmp1}\ of \\ 0 :\ & foreign\ \textbf{llvm}\ \text{move}(0,?ret) \\ 1 :\ & foreign\ \textbf{llvm}\ \text{urem}(a,b,?b') \\ & \quad gcd(a,b',?ret) \end{aligned}$$

Figure 5.4: Comparing the clause form of GCD (left) with its actual LPVM implementation (right)

body primitives and will end up calling itself for iteration. The second will contain the body primitives which comes after the loop. The clause body that the loop was a statement in originally will end with a call to the first generated procedure. The first procedure, in turn, will contain a breaking conditional or guard to call the second generated procedure.

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 5.3. 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*. The original loop statement will now just be a call to *next1*. Since LPVM procedures support multiple output, we can have the same scoped variables which came out of the loop block as the outputs in *next1*.

Figure 5.4 shows the implemented LPVM version of the *gcd* function we saw earlier. On the left is the abstract form representation, and on the right is how it looks like in the actual implementation. The optimisation and analysis methods given in Gange et al. (2015) are still possible for the above implemented structure. The LPVM structure also does not define a fixed instruction set, in fact its instruction set is being provided by LLVM. LPVM exists as an important glue structure between Wybe and machine instructions in the form of LLVM.

Chapter 6

Transforming Wybe to LPVM

The Wybe syntax tree is systematically transformed into the LPVM IR data structure. In this process it undergoes *flattening*, *type checking*, *un-branching*, and a final *clause generation* to obtain a structure similar to Figure 5.2. 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 level provides a lot of benefits which are apparent in the incremental features of the compiler discussed in Chapter 7.

The type which stores the implementation of a Wybe procedure, in source and LPVM form, is shown in Figure 6.2. The complete procedure definition *ProcDef* will also contain other information about the callers, visible types, and more. A procedure can have multiple implementations, each of which corresponds to a different *Clause* of the procedure. Initially a *ProcImpln* will be built from the constructor *ProcDefSrc*, indicating source language form. On completing transformation to LPVM, the same type will have the constructor *ProcDefPrim*. The *Proto* and *Body* are similar to the constructors in Figure 5.2.

The compiler implementation keeps the pipeline modular. While the module implementations are stored in a flat *List* data structure, it is possible to generate a module dependency graph (transitive closure of dependencies) given a module name. The implementation for any given mod-

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

Figure 6.1: Normalisation of Wybe semantic structures to Procedures

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

Figure 6.2: The Procedure Implementation Algebraic Data Type

ule name can be obtained easily as long as it exists in the above list. The module currently under compilation will be at the head of this list. This also makes loading implementations from an external source and removing module implementations from the pipeline very easy, assisting in the implementation of the incremental features discussed in Chapter 7.

Since LPVM procedure signatures should also present a pure interface like Wybe, Wybe *resources* are passed as a new extra parameter to the procedure. LPVM has no primitive syntax for Wybe *resources*. 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. An example of this is shown in Figure 6.1. This would mean that the LPVM procedure takes in an *io* resource, 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 memory location.

While passes of the compiler during transformation can be discussed independently, they are not truly sequential. Some passes and transformations happen at the same time and are dependent on each other. Nevertheless, we can talk about the important transformations that occur on the Wybe source code as its LPVM form is built.

6.0.1 Flattening Pass

During compilation every top level structure 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. A procedure body contains a sequence of statements which build up the outputs. Hence all expressions forms have to be flattened to appear as statements too. Some common conversions are shown in Figure 6.1. The top level statements in the module are put into a separate nameless procedure. The code generation phase will later make this the *main* function of the module.

Since LPVM primitives are in the form of procedure calls, all normalised Wybe statements are gradually reduced to procedure calls. These primitive procedure calls can be *local* calls to other procedures in the module or imported Wybe modules (fully qualified procedure names), or they can be

$$\begin{array}{lll}
type\ bool\ = public\ false\ |true & \rightarrow & \mathbf{false}(?t : bool) \\
 & & \mathbf{true}(?t : bool) \\
 & & =(?out : bool, in : bool) \\
 & & =(out : bool, ?in : bool) \\
 & & =(x : bool, y : bool, ?z : bool)
\end{array}$$

Figure 6.3: Normalisation: generation of procedures to provide an interface to a user type

foreign calls. Foreign calls reference procedures or instructions which have to be addressed later by the code generation phase. This is usually done by linking in an external object file or using LLVM instructions. For example, the Wybe standard library defines a procedure *println* whose body statements are foreign calls to C *stdio printf*. A shared C library will be linked with the standard library to resolve these calls to access system IO. 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.

The result of the flattening pass is a very normalised structure of only procedures. Apart from the above conversions, the normalisation which happens along with flattening also sets up the *Module* type which will hold the internal representation of a Wybe module in the compiler. Sub-module definitions and type definitions which occur in the same source module are separated into separate *Module* types and their module names are qualified with the parent module name.

Normalisation will also expose the defined types and public procedure names in the module, and add default implementations of assignment and comparison procedures for user defined types. Note that the = symbol is used for denoting assignment and comparison here. They are differently evaluated depending on the modes and number of parameters. here A user introduced type constructor is normalised to a new procedure. For example, the *ADT* definition for *bool* has two constructors, *true* and *false*, which create the procedures shown in Figure 6.3.

Normalisation in a compiler ensures that we will be operating with a well defined and closed set of primitives hereafter. Addition of new syntax features can be independently done as long as they can be normalised to the same set of primitives. This greatly simplifies compiler construction.

6.0.2 Type Checking Pass

Wybe, and LPVM, are statically typed. The type checking pass is essential to enforce this strictness and robustness of the type system. It also connects polymorphic procedure calls to the correct procedures by matching call signatures with procedure prototypes. Explicit typing is also only enforced on Wybe function and procedure prototypes. The variables in the body will have inferred types based on the input and output types.

Every variable name in the AST will be annotated with an inferred type at the end of this pass. As mentioned above, every type in Wybe is defined in its own module which contains all the procedures used to interact with that type. Hence, this pass also connects type names to the modules that provide the definition for that type. This is required even for standard types like *int* since it can be provided by a non standard library just as easily.

Polymorphic calls are resolved by looking into imported modules in reverse for a matching procedure prototype. Since every module auto imports the *wybe* module, the type modules defined in it will be in the imported module list for any particular module. For example, the equality procedure ′=′, 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, like the call *proc call =(a, b, ?c)*, will be annotated as *proc call wybe.int.=(a:wybe.int, b:wybe.int, ?c:wybe.int)*.

At the end of a successful type checking pass, 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 Chapter 5.

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. Procedure calls reflect the clause they are trying to actually call with a suffixed index. For example a call to *foo*<2> is referencing its third clause in order (starting from 0).

After this pass an entire module is just a list of clauses. This representation will be passed through further optimisation passes and finally LLVM code generation. This clausal form is also independent of the Wybe source. Given this form directly, we can still optimise it and generate the same code with it as if we were given the Wybe form. This is an inherent property of an Intermediate Representation and that's why it is so useful to have in compiler construction.

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 replaces most of these calls and avoids procedure call overheads. The decision to inline is based on a cost vs benefit analysis. This heuristic takes into consideration the number of callers to that procedure, it's valid arguments, and body size in number of statements.

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 even 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. It is only these variables which will have to be renamed to fit in the procedure body it is being pasted into. This form of In-lining is supported by 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 the body into a single LLVM instruction.

Chapter 7

Wybemk, Compiler and Build System

Wybemk is the incremental compiler for Wybe source code and a Make-like build system combined together in one executable. It doesn't depend on a Makefile or header/interface files to compute dependency graphs. 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 *bitcod*e files, or a final linked executable. The object files and *bitcode* files that Wybemk builds are a little different than what other utilities create, but still recognised as an ordinary object file. They have embedded information that assists a future compilation process. It is this embedding that allows Wybemk to be an incremental and work-saving compiler.

Like Make, if an object file for a corresponding source module is found to be newer in its modification date-time, the object file is not re-built. But by not re-building a target object file, the LPVM form and analysis for that module is also skipped. This is acceptable for only intra-module optimisation phases since the final optimised object code will be the same. But we might be missing a lot of inter module optimisation opportunities and LPVM in-lining that other dependant modules can reap benefits from. Object files normally 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 in-lining 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 this decision. Thus, 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 into the compilation pipeline.

The fairly limited structure of LPVM makes serialising and embedding the resulting byte structure into a object file practically feasible. If we were spending more time serialising and doing this IO operation, it would not be an acceptable trade off. Instead of the LPVM form, we could have also stored the parse tree. But a parse tree has a wider form and is redundant with the source code file. With storing the parse tree we are only skipping the work the parser does and would still 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, the Wybemk compiler can do the same embedding with LLVM bitcode files. LLVM bitcode files can be treated as architecture neutral. Since we use a LLVM compiler as a final stage, we can use bitcode files as a replacement for architecture dependent object files.

In Section 7.1 we discuss our method of embedding a byte string in an object file and bitcode file. After which, we discuss how storing different types of data exposes different approaches to being incremental and work-saving in Section 7.2.

7.1 Storing structures in Object files

Object files store relocatable object code which is the compiled code generated by the final LLVM compiler in the Wybemk compilation pipeline. Even though different architectures have their own specification of an object file format, they are modelled around the same basic structure. An object files defines segments, which are mapped as memory segments during *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 in the file), which do not get mapped to memory. These are zero address segments. Using such a segment we can attempt storage of some useful serialised meta-data in the object file without bloating the linked executable.

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. In this case, an ABI describes the byte ordering and their meaning for the operating system. The extra embedded bytes should not interfere in the usual semantics of the object file. The object file should 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 its total byte size.

7.1.1 Mach-O Object File Format

Quite simply, an object file is a long sequential byte string sequence. Every byte is semantically important. 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 the *little endian* format. The magic number constant determines what kind of ABI the rest of the bytes of the file follow and their *endianness*. 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 mostly interested in *Mach-O* object files.

The header bytes of the identified structure will give the number of *load commands* in the file. The *load commands* are the segments which get loaded to the main memory during execution. The bytes of a *load command* will provide the name of its segment and an offset pointer to the location of its data in the file. Each segment can also contain multiple sections. A complete object file has several important pre-defined segments and sections which should not be touched by us to prevent altering the original object file. Wybemk creates a completely new segment called *__LPVM* and a section in it called *__lpvm*, following naming conventions. This adds a new load command and increments the number of load commands the object file has. Once the offsets are correctly determined and bytes describing our new load command put in its right place, we can insert our byte-string at that offset. We will have to alter the offsets of the subsequent load commands. To make it easier, we rather have our new load command data at the end of the object file. This byte-string will be the encoded serialised form of the data type we want to embed.

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* machines, 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.

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

7.1.2 Bitcode Wrapper

LLVM IR can be put in *bitcode* files[2]using LLVM tools. These are binary representations of the LLVM IR. These files are identified by a magic number, similar to a *Mach-O* file. Other LLVM tools can easily generate these formats and even compile them to an 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 file differs from an ordinary *bitcode* file in its initial magic number. This wrapper format specifies header bytes which should give the 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 the LPVM information which usually goes into the object file, in a serialised form.

7.2 Incremental Compilation

Wybe, as a new programming language, has a goal to be useful for large scale projects. Thus, Wybemk should be as efficient as possible in its compilation process. In bigger projects, a large number of modules are involved in a single build. Having a heuristic to selectively compile and yet obtain the same working build as a full compile massively reduces turn around time over a period (Adams et al., 1994). Having all the modules compile again is a waste of time.

Another method to achieve efficiency in building is to be incremental. If the compiler can operate at a granularity lower than the entire source file, it can save and reload object code of smaller units of the source file (Cooper and Wise, 1997). In Wybemk we can operate at the granularity of LPVM procedures. Since Wybe functions and procedures all get compiled to a LPVM procedure form, we can re-use LPVM forms of Wybe structures we have seen before. To do this, we identify certain key stages in the compilation pipeline which can act as save and restore checkpoints. The saving is done using object file (or bitcode file) embedding, 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 the 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.

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

7.2.1 Module level reloading

Wybemk compiler behaves like GNU Make but it does not depend on Makefiles or separate interface/header files for modules. For a given target, it is able to determine the transitive closure of its dependencies. Inferring the dependency graph is done through parsing the top level *use module* statements and sub-module declarations. There is a restriction here that modules and their corresponding object files are matched by name and location only. So for a module, Wybemk wants to find its object file (if it is built) in its search path.

The Wybemk compilation pipeline is composable. For a given list of source modules, it will build up the LPVM form for only those modules which don't have an LPVM form in the pipeline. So a LPVM form can come from anywhere. This LPVM form is a part of what we embed in an object file on a successful build. During compilation, modules which have a newer object file do not undergo re-compilation. However, the LPVM form stored in the byte string embedded in that object file is deserialised and placed in the pipeline. The compilation will continue transparently.

As mentioned above, one of the derived utilities of having the LPVM form for every module at hand is in-lining. LPVM level in-lining is also discussed in Chapter 6. For example the *int* module in the Wybe standard library module (Figure 4.1) 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. The standard library is only compiled once in it's entire lifetime. In-lining at LPVM level provides these small but noticeable benefits.

We want to embed the minimum set of data we will need for the next compilation. In our implementation we are going to be serialising an *Abstract Data Type*. This ADT is the subset of the type which holds information on an entire *Module*. The complete *Module* type describes the exported types, procedure implementations, sub-module names, dependency information, LLVM implementation, and more, for a module. The serialised subset will remove local information and just preserve the fields which other modules might need to read. For example, the LLVM implementation is already present, in essence, in the *TEXT* segment of the object file in object code form. If a source module contains multiple sub-modules, the serialisation is done on a list of *Modules* instead in a fixed order.

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 inter-module optimisation. Serialising the LPVM form instead of the source code form already saves a lot of space, so extra space saving heuristics are a bonus. The exportable *Module* subset is packed as a byte string. A LPVM primitive is made up of either of two constructors (Figure 5.2): *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 should be used while decoding.

Reloading of the entire LPVM module is done when the object file has a newer modification time. This is an assurance that the LPVM form reloaded is representative of the correct source code. This also means the entire source module or a build of several modules should successfully compile at least once. After which, we get time savings on changes to individual modules in subsequent builds. This is how Wybemk implements its work-saving features. The next stage, being incremental, is designed to work when the object file can be older than the source. This is discussed in the next subsection.

7.2.2 Incrementality through Storing Hashes of Stages

Given a newer source file and it's last built object file we want to build a new object file which contains parts from the old object file which was not affected by the source code changes. In this scenario, simply loading the entire LPVM form from the object (shown above) is not correct. 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 conservatively as having numerous checkpoints will start slowing down the compiler instead of saving time. Reloading of old work should also be done carefully as dependencies in the current compilation process might not be the same as last time.

The level of granularity we work at must be able to generate complete transitive closures of dependencies (Cooper and Wise, 1997). The Wybe top-level contains only functions and procedures (top level statements are placed in another procedure of their own). Both these functions and procedures are flattened to look like procedures, as discussed in Chapter 6. The *ProcDef* type in the compiler (Figure 6.2) contains information on the callers of a procedure, and the procedures which in-line it. This information can help us compute the dependent set of procedures for any given procedure. Procedures belonging to that dependent set will be affected by a change in the root procedure. It should be noted that the dependent set is local to the module. It is not possible to determine affected procedures across module boundaries. A changed procedure will mark itself changed. Procedures from other modules can check this mark to determine their own condition. The dependency graph of modules is compiled depth first so an external procedure from another module would already have a compiled form.

We can define the Wybemk compilation pipeline as being a transformation from Wybe source code to LPVM, and then a transformation from LPVM to object code. Let's assume that LLVM IR and object code generation are part of a single atomic step. During the first transformation, from Wybe to LPVM, we first arrive at an elementary form of LPVM and then run optimisation passes on it until we arrive at a final LPVM form. A Wybe procedure is transformed to one or more LPVM procedures as shown in Chapter 6. The final LPVM form for each procedure already exists in the object file. If we have the relation between a Wybe procedure name and the names of the LPVM procedures it is transformed to, we can load the LPVM procedure forms from the object file without doing the transformation and optimisation passes. Note that this would require the LPVM forms of even private procedures to exist in the object file. We have to revoke the decision we made earlier to store only public procedures since they are the ones which get inlined.

To determine whether we should re-load the LPVM form of a Wybe procedure or do the transformation we us hash comparisons. At a checkpoint, we run a hash function on the current form at hand and compare it with the hash stored in the object file (for the same stage). The hash stored in the object file is from the last time that form was actually compiled. These hashes are stored in a *Map* in the subset of the *Module* type that gets serialised and embedded in the object file (shown above). The hash function we are using is the old popular checksum algorithm *md5*. We are ignoring collisions and speed for now to reach a working implementation first. There is definitely future work involved in choosing the correct algorithm

An early and elementary stage of compilation at which we can use hash comparisons is the post-parsing stage. After parsing we can compute the hash of the *Abstract Syntax Tree* type generated by the parser. The hash of the previous AST (from a previous compilation) is loaded from the object file. By comparing these two hashes we make a decision on whether to proceed with recompilation or not. For source code edits involving changing or adding comments and white-space, the complete 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, just like the case when the object file was newer. 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.

After the parse tree check, we can begin considering Wybe procedures individually. We still wait until the flattening stage so that we are dealing with a single normalised procedure form. The hash function operates on the *ProcDefSrc* type. In the object file we store the hash of the same form from a last complete compilation. As mentioned above, for every procedure P_i we also have a set of procedures which are dependent on the current interface of P_i. If a procedure has changed, it will be re-compiled until LPVM. This will also trigger a recompilation for the procedures in the dependent set. It can be observed that there can be a set of procedures which have not changed and aren't in the dependent set of a changed procedure. It is safe to reload the LPVM form for these procedures from the object file into the compilation pipeline. Not recompiling this set is the first elementary gain we are accomplishing.

The dependent set for a procedure P_i can be divided up into two subsets. The first subset (Q) include procedures which in-line P_i in their bodies and the second subset (S) just make a call to P_i. A procedure S_j only depends on some version of the prototype/signature of P_i. If we can determine that the prototype of the changed procedure P_i has not changed, then we can remove the subset S from the dependent set of P_i. However, a procedure Q_j will always be re-compiled for a change in P_i.

There are also cases where changing a variable name uniformly throughout a procedure body would not change it's LPVM form since LPVM will be generating it's own unique variable names (like SSA). These kind of edits don't change the semantics of the procedure and re-compilation is not needed. To derive gains from this observation, we plan to next track relations between the LPVM form and object code. Implementing this would require having the ability to load and change object code directly from the *TEXT* segment of an object file instead of our custom segment.

There are many stages and scenarios like the ones above where we can smartly avoid recompilation. In the beginning, the goal is to incorporate the simple scenarios which are absolute in its resolution of a recompilation need. In-lining across modules causes a domino effect of recompilation of multiple procedures. For now, even if we are making small savings for some compilation processes, we may be making bigger savings on smaller edits. As long as we are able to achieve an overall gain in turn around time over multiple builds, we are still doing better than a non-incremental compiler since we are performing less compilation passes. The embedded information in the object file is loaded in the beginning, which has the side effect of increasing the memory consumption of the compilation step but limits IO actions. With modern machines, this trade off is acceptable.

7.3 Examples of Gains

The best observable result obtained with the module level reloading is the use of Wybe standard library. The entire standard library lives in one file, *wybe.wybe*. All the procedures in the types defined in it are in-linable, and in fact do expect to get in-lined. With a fresh redistribution of the standard library module, a compilation of an arbitrary module *foo.wybe* will also build the object file *wybe.o*. This happens because the compiler adds the dependency on the module *wybe* implicitly for every compilation target. In building *wybe.o*, the LPVM and LLVM forms are created from the source code over numerous passes. The LLVM form generates the object code, while the LPVM form gets serialised and stored in the object file. Hereafter, every compilation done by Wybemk will never build *wybe.o*. Instead, the pre-built LPVM form of all the *Modules* in it will be loaded and used. Since every call to the standard library is mostly in-lined, having the LPVM form proves beneficial. In any particular build, this also happens only once even though every module will depend on module *wybe*. This is due to the modular compilation pipeline.

Chapter 8

Code Generation to LLVM

We do not generate machine code directly from the LPVM IR form. Instead we are hooking into the LLVM framework (Lattner, 2002) so that we are able to compile Wybe programs on multiple architectures without additional effort. In this chapter this transformation from LPVM to the LLVM IR form is presented. Without the LLVM project we would have to write code generators for every popular architecture and more. LLVM is a popular open source project with a lot of collective effort being poured into it. A lot of older mature compilers (and new) are also being re-targeted to LLVM IR, and the efforts taken previously in making the machine code generator efficient is being poured into LLVM. We feel it's worthwhile to do the same for Wybe.

The LLVM IR has an abstract structure of an imperative program. It is a SSA based form. Earlier we talked about how LPVM IR solves the drawbacks of the SSA naming scheme and its virtual functions, and yet we are ultimately targeting a SSA based IR. We do this based on the observation 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 its LLVM IR transformation. In its final stage, basic LPVM *clause* bodies end with either building it's outputs or a fork. This fork does not merge back into another block. Instead, every other basic body in the fork will either terminate finishing its work or will end in a call to another procedure. With no converging bodies, there is no need of using a *φ-function* in LLVM. 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 LLVM IR easily. We then use the system linker to link these object files. For accessing instructions not provided by LLVM, we link in a custom shared C library. Currently this library allows access to a few standard IO functions and the *stdlib* of C. 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 file versions. Using this library we can provide a stronger support for compound types which need access to heap memory, as discussed in section 8.2. Finally we discuss how Wybemk does its dependency linking and deals with top level code for every dependency in section 8.3.

8.1 Transforming LPVM to LLVM

Since LLVM is based on an imperative form, we have to transform each LPVM procedure back to an imperative function. An imperative function and a procedure are somewhat similar in that they both have a sequence of imperative statements as their body. Wybe and LPVM IR support procedures with multiple outputs, but imperative functions do not do so by default. We need the flexibility of multiple outputs reflected in the LLVM IR as well. Having multiple outputs is 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. This aggregate structure type is allocated on the stack of the function instead of the heap, making it similar to a local variable. The nameless procedure which holds the top level statements of the module is transformed to a *module.main* function. The prefixed module name is used to differentiate from *main* functions of other modules.

The LLVM IR looks like a C program. At the top level, it contains *declare* statements (to refer to an extern function or some global constant) and *functions*. A function has a return type and parameters defined exactly like a C function. Void functions are acceptable. The body of a function is composed of basic blocks. A basic block ends with a *terminator* statement which is either a return statement (*ret*) or a branching statement (*br*). A basic block will branch into alternate basic blocks. Eventually the basic blocks converge, possibly to a block with a *φ-function*. Statements in a basic block are *instruction* calls. LLVM provides numerous predefined *instructions* for which it can generate machine code. They model *instructions* of an assembly language. Examples of some *instructions* are: *add*, *sub*, *icmp*, *store* .etc. LLVM uses pointers and indexing to access memory locations in the heap. Allocation of heap memory is left up to the programmer to implement. The correctness of the bounds in indexing is also left to the programmer. There are no exceptions. LLVM provides a few compound data structures like the structure, array, and vector as well. Being a high level IR it assumes an abstraction of infinite registers. The LLVM framework also provides numerous tools and utilities to compile, assemble, disassemble, link, optimise, .etc. LLVM IR modules.

Referring to the structure in Figure 5.2, we know that a LPVM primitive statement is either a *local* procedure call or a *foreign* call. We mentioned earlier that these calls are meant as a directive for the compiler to generate specific machine code. 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 LLVM IR these are called by first declaring this function name as an *extern* and then generating a *call* instruction for 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. Foreign calls to *lpvm* are special functions whose implementation can be provided by anyone. Currently we have memory allocation, access, mutation functions in this group, and we use *C* functions to provide the implementation. This is discussed in more detail in section 8.2.

The *GCD* function for which we showed the LPVM transformation earlier, is now shown being translated to the LLVM IR in Figure 8.1. The two clause branches spawned off the condition on goal *b* end in a *return* statement 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 direct 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).

We have seen that 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 mentioning certain flags in the generated *call* instruction and ensuring that the correct calling convention is used. We also have to tell the LLVM code generator to optimise these marked calls with the *-tailcallopt* optimisation flag. Finally, the function call must be an actual tail call. Since the block has to return a value in its terminating statement, a function call can't actually be in the tail position of a function. However, if a return instruction follows the actual call and returns the value returned by the function or returns void, that call is considered to be a tail call. In Figure 8.1 the recursive call to *gcd* is an example of a LLVM tail call.

The LLVM target independent code generator provides numerous optimisation passes. While generating code, these passes can be specified by their name and LLVM will run them in order before generating assembly code. A curated set of passes selected by the LLVM team is also provided under a single compound pass. All of these passes can be run during code generation or on an LLVM IR *bitcode* file using the *llvm-opt* utility. Currently in our implementation we are using an elementary set of passes so that we

```
$\mathbf{gcd}(a, b, ?ret) \rightarrow$
            $foreign$ $\mathbf{llvm}$ icmp ne$(b, 0, ?tmp1)$
            $case$ tmp1 $of$
        0 : $foreign$ $\mathbf{llvm}$ move$(0, ?ret)$
        1 : $foreign$ $\mathbf{llvm}$ 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: Comparison of the LPVM form of *GCD* to its LLVM form

can focus on LPVM more. For every new pass added to the Wybemk pass set, we have to ensure that it is not redoing the analysis LPVM has already done. Similarly, for every pass LPVM wants to implement, we check if LLVM already provides some equivalent pass.

8.2 Code generation for Wybe types with LLVM

LLVM provides access to a basic integer type of arbitrary bit precision and the float type. The syntax *i32* represents an integer with 32 bit precision in LLVM. In Wybe, we are considering an *int* to be of the machine word size, which is commonly *i32* or *i64*. However, the bit size of the standard *int* type can be specified through Wybe type syntax as shown in Chapter 4. Using the basic LLVM types we can model all of Wybe's basic types like *int*, *float*, *bool*, *char* .etc.

Compound types and *Abstract Data Types* require allocation on the heap. 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* procedure calls accordingly. As mentioned in Chapter 6, a Wybe type definition results in the generation of *setter* and *getter* procedures for members of that type. One such generation is shown in Figure 8.2 for an ADT *position* which represents a geometric Cartesian coordinate. Internally, for LLVM, the type *position* will be represented as a pointer to a word size integer (i64* or i32*). A *position* type pointer will point to the location where its members are located on the heap. The *position* type constructor takes two integers to build itself. Wybemk determines that this needs a total of 16 bytes allocated on the heap, 8 for each member (assuming an int is defined as *i64* here). In the two clauses shown for the constructor procedure for *position*, clause 0 packs up two integers into a *position* type and clause 1 unpacks two integers from a *position* type. The *mutate* and *access* procedure calls pass a constant *int* as the offset index.

On the side of LLVM, the instructions *getelementptr*, *store*, *load* are used to index a pointer, store data using that index, and load data from that index respectively. The *foreign* calls shown above all translate to some combination of these three LLVM instructions, along with *casting* instructions. In a way, *alloc*, *mutate*, *access* are LPVM instructions, with their machine code provided in LLVM assembly.

By default, LLVM only provides heap indexing instructions. The allocation has to be implemented by the programmer. An elementary way to do this is to use LLVM function calls to C *stdlib*, for which an object file providing access to this library has to be linked in later. By using calls to *malloc* and *free*, we already have a simple memory interface set up. These calls return a *void* pointer which can be *bitcasted* to the required pointer type using LLVM *instructions*. The LLVM instruction *getelementptr* can then index

Wybe

$$\mathit{public}\ \mathit{type}\ \mathbf{position} = \mathit{public}\ \mathbf{position}(x : \mathit{int}, y : \mathit{int})\mathit{end}$$

LPVM

$0 : \mathbf{position}(x : \mathit{int}, y : \mathit{int}, ?\mathit{out} : \mathit{position}) :$

$\quad \mathit{foreign}\ \mathrm{lpvm}\ \mathbf{alloc}(16 : \mathit{int}, ?\mathit{temp}\#0 : \mathit{position})$

$\quad \mathit{foreign}\ \mathrm{lpvm}\ \mathbf{mutate}(\mathit{temp}\#0 : \mathit{position}, ?\mathit{temp}\#1 : \mathit{position}, 0 : \mathit{int}, x : \mathit{int})$

$\quad \mathit{foreign}\ \mathrm{lpvm}\ \mathbf{mutate}(\mathit{temp}\#1 : \mathit{position}, ?\mathit{temp}\#2 : \mathit{position}, 1 : \mathit{int}, y : \mathit{int})$

$\quad \mathit{foreign}\ \mathrm{llvm}\ \mathbf{move}(\mathit{temp}\#2 : \mathit{position}, ?\mathit{out} : \mathit{position})$

$1 : \mathbf{position}(?x : \mathit{int}, ?y : \mathit{int}, \mathit{out} : \mathit{position}) :$

$\quad \mathit{foreign}\ \mathrm{lpvm}\ \mathbf{access}(\mathit{out} : \mathit{position}, 0 : \mathit{int}, ?x : \mathit{int})$

$\quad \mathit{foreign}\ \mathrm{lpvm}\ \mathbf{access}(\mathit{out} : \mathit{position}, 1 : \mathit{int}, ?y : \mathit{int})$

Figure 8.2: LPVM Procedures generated for an Abstract Data Type

the heap allocated memory. With this solution, the garbage collection, or calling *free* at the correct places, becomes a big task. Since we also want to make headway into providing garbage collection in the compiler, it would be better to have a more intelligent memory interface.

Without providing a complete solution, we can still access dynamic garbage collection by making use of the The Boehm-Demers-Weiser conservative garbage collector (Boehm, 2004). 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 GC to automatically collect memory. We don't have to worry about making correct calls to *free* anymore. To setup 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.

8.3 Linking Dependencies

With LLVM we can link a transitive closure of a modules' dependencies in two ways: build object files for each module and use the system linker, or use the *llvm-link* utility to link LLVM IR forms of each module together into one file and build a single object file of that. Since we are exploiting the structure of object files to support our incremental features, we are inclined to follow the first approach.

In the current implementation we are invoking the system *ld* linker when we have the object files ready. When the linker is given multiple object files to link in an executable, it expects only one of the passed object files to define the *main* symbol. Currently modules with a non empty set of top level statements have a *module.main* procedure defined, which is the entry point for that *module*. This procedure should be executed when its module is imported into another or when it is the target executable. This procedure is transformed to a function of the same name in LLVM IR. During linking, for every module in the dependency graph, all of these local *module.main* function bodies should be concatenated into a single body, in depth first order of import, and made available under the *main* symbol of the executable object code. For example, let's assume we are building an executable target foo and the dependency chain looks like:

$$foo \leftarrow aaa, bbb \qquad aaa \leftarrow ccc, ddd \qquad bbb \leftarrow ddd$$

The *module.main* functions ($aaa.main()$, $bbb.main()$...) are concatenated in the order: ccc, ddd, aaa, bbb, foo. Notice how $ddd.main$ wasn't considered twice even though it would appear twice in the traversal. There is no need to run the initialisation code for that module again. This concatenated body is put into a new temporary object file under the function name *main* and added to the linking list so that *ld* can find the *main* symbol it needs.

As mentioned above, apart form the Wybe object files, we are also linking in a C shared library. Since Wybemk is supposed to be a complete build system solution for Wybe, it should also be able to link in dynamic system libraries built with other compilers. A Wybe source module should be able to request linking of some external object file. This is up-coming in the future versions. Such a construct would also replace the need for the current C shared library we use. Instead, the usage of some C header/library can be explicitly specified, and Wybemk will search the system library paths for it. The goal is to keep the compiler and standard library independent. This is similar to how many languages provide their standard library separately from the compiler. Our independence also extends to the implementation of primitive types in the compiler, since the Wybe source file of the standard library can state the size of *ints* in bits.

Chapter 9

Conclusion and Future Work

The Wybemk compiler provides the implementation that was talked about as part of future work in Gange et al. (2015). With LLVM, running complete Wybe programs is easier. As a build system solution, Wybemk makes a lot of opinionated decisions which are derived from observing the good parts of various other build systems and compilers. Current and future ideas for extending the implementation will continue to push Wybe to a level where it can compete with new modern languages.

The Wybe programming language still has a lot of syntax changes and new features coming along in its future pipeline. It's a new language and complete syntax re-works are common. The changes to Wybe syntax naturally changes a lot of existing LPVM structures, which in turn have to be dealt with in LLVM code generation. Work on adding more LPVM analyses and optimisation passes is also underway. We also want to add a more complete static garbage collection option to Wybe and extend the typing implementation to be more space effecient. All modern languages feature memory robustness and safety, and Wybe will have to compete with them

The build system of the Wybemk compiler is consistently being made more incremental. Some of the unimplemented possibilities have been discussed in Chapter 7. The hash function used for comparing *checksums* can also be chosen more intelligently.

Along with newer approaches to be more time-saving, we also want to extend our compiler to be more cross platform. We have shown how we embed information into *Mach-O* object files for OS X only. Using the cross platform LLVM *bitcode* files is still an option. Currently Wybemk can build an object archive target for a folder of Wybe source files. This is the first step towards packages in the build system. However we do not embed any information into the archive files yet. Our use of normal object files makes it possible to link Wybe with object files built by other compilers. Since Wybemk is supposed to be a complete solution for compiling and building, it should be intelligent enough to know which system libraries to link in for any use-case. For example, Wybe should be able to explicitly say that it wants to be linked with some C GUI library on the system.

While we want to exploit the new logic IR, LPVM, as much as possible and use LLVM for code generation, we don't want to duplicate LLVM optimisation methods. The LLVM back-end framework provides numerous optimisation passes for which a lot of work has already been put it. We don't want to shy away from these. Testing the effects and incorporating more advanced LLVM optimisation in tandem with LPVM passes are planned for future work.

Bibliography

Adams, R., Tichy, W., and Weinert, A. (1994). The cost of selective recompilation and environment processing. *ACM Trans. Softw. Eng. Methodol.*, 3(1):3–28.

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.

Ananian, C. Scott (Clifford Scott), . (2001). The static single information form.

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

Boehm, H.-J. (2004). The boehm-demers-weiser conservative garbage collector. *HP Labs*.

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.

Groff, J. and Lattner, C. (2015). Swift intermediate language. `http://blog.rust-lang.org/2016/04/19/MIR.html`. A high level IR to complement LLVM.

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`.

Matsakis, N. (2016). Introducing mir. `http://blog.rust-lang.org/2016/04/19/MIR.html`. The Rust Programming Language Blog.

Ottenstein, K. J., Ballancel, 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.

Schwanke, R. W. and Kaiser, G. E. (1988). Smarter recompilation. *ACM Trans. Program. Lang. Syst.*, 10(4):627–632.

Tichy, W. F. (1986). Smart recompilation. *ACM Trans. Program. Lang. Syst.*, 8(3):273–291.