Research Proposal for Higher order programming in Wybe

by

James Barnes
Student Number: 820946

Supervisor: Peter Schachte

Word count: 5977

A research proposal submitted in parital fulfillment for the
degree of Master of Computer Science

in the
Department of Computing and Information Systems
Melbourne School of Engineering & IT
THE UNIVERSITY OF MELBOURNE

October 2021

THE UNIVERSITY OF MELBOURNE

Abstract

Department of Computing and Information Systems
Melbourne School of Engineering & IT

Master of Computer Science

by James Barnes
Student Number: 820946

Higher-order programming allows for a greater level of abstraction over first-order programming. Allowing for procedures to be passed into or out of a procedure is the hallmark of this programming style. An emerging programming language, Wybe, currently does not support higher order programming, nor does the intermediate representation used internally by the Wybe compiler, logic programming virtual machine (LPVM). In this research, we propose an extension to LPVM, and further to Wybe, to support higher order programming. Wybe presents novel language features, which together with higher order programming, provide a novel language implementation space.

Acknowledgements

I would like to acknowledge the people of the Kulin nations, the traditional custodians of the lands on which this work was produced, particularly the Wurundjeri and Wathaurong people, whose lands I both work and live on. I pay my respects to the elders, past, present, and future.

Contents

Chapter 1

Introduction

This research proposal is primarily concerned with an extension to an emerging programming language, Wybe [1] (pronounced ['wibə], Wee-Buh), to extend the language to support higher order programming. Wybe is intended to be a language that teaches principled programming practices from learning through to practice.

A defining feature of Wybe is a strong emphasis on interface integrity. Interface integrity refers to the property that all possible effects of a procedure are known from the interface alone, without the requirement to delve into finer-grained implementation details.

The type system of Wybe currently supports overloading based on type and mode of the arguments of a given procedure. Overloading refers to the ability to use the same name for multiple different implementations. Overloading of modes allows procedures to be executed "in reverse", with arguments allowed to be input or output, depending on how the procedure is called. Modes are typical of logic languages such as Prolog [2], however are presented differently in Wybe by allowing for explicit overloading of modes. Wybe also supports generic, or polymorphic, types.

```
def add( x:int,  y:int, ?z:int) { ?z = x + y }
def add( x:int, ?y:int,  z:int) { ?y = z - x }
def add(?x:int,  y:int,  z:int) { ?x = z - y }

?a = 1; ?b = 2
add(a, b, ?c) # binds c to a + b
add(a, ?d, c) # binds d to c - a
```

Figure 1.1: An example Wybe program showing the use of multiple modes. Outputs are marked with a preceding `?`, and inputs are unmarked.

Wybe employs a novel message passing mechanism called resources. A resource allows data to be passed by name rather than position. They are intended to be used as a parameter that is unique in a computation and also widely used, being threaded between resourceful procedure calls automatically. The quintessential example of a resource in Wybe is the `io` resource. `io` represents the state of the outside world, providing a declarative interface for input/output operations. Each call to a procedure that uses the `io` resource ensures that a change to the "state" of the outside world is represented as a change in the `io` resource. Other examples of resources are the program arguments vector and count, `argv` and `argc`, respectively.

```
def print_sum(a:int, b:int) use !io {
    ?c = a + b
    !print(a); !print(" + "); !print(b)
    !print(" = "); !println(c)
}

!print_sum(a, b)
```

FIGURE 1.2: An example Wybe program showing use of the `io` resource. Calls to resourceful procedures, such as `print` are marked by a preceding `!`.

Unlike many other languages (such as Haskell [3], and Java [4]), Wybe does not support higher order functions or procedures. That is, functions or procedures are only able to pass parameters that are first order. Higher order functions allow for code to be written at a greater level of abstraction than first order code and allows for code to be written in a distinct and principled manner that is not possible with first order code. This limits the utility and expressiveness of the Wybe language.

The Wybe compiler, `wybemk`, employs a novel intermediate representation for its initial transformation and analyses phase, logic programming virtual machine (LPVM) [5]. Despite its name, LPVM is not a virtual machine, but an intermediate representation that makes use of restricted Horn clauses, similar to that seen in logic programming languages such as Prolog [2] and Mercury [6]. With the restrictions to Horn clauses, LPVM has been shown to be an efficient intermediate representation that solves many issues in other intermediate representations seen previously.

1.1 Research Questions

The intention of this project is to explore the design and implementation of higher order types in both the LPVM intermediate representation and the Wybe programming language. Further, to ensure that higher order types in Wybe and LPVM are able to be used effectively and provide benefit to a programmer, the extended language and intermediate representation will be compared with the current implementation of both.

Motivating this proposed research, three key questions are as follows:

- How can the existing LPVM implementation be extended to efficiently incorporate higher order types?
- Can the the existing features of the Wybe language, namely resources and different properties of procedures (such as partial procedures), be extended to support higher order types?
- Do these extensions allow for adequate execution when compared to the existing Wybe language and LPVM intermediate representation?

1.2 Document Overview

The structure of this research proposal is as follows.

Firstly, a literature review (Chapter 2) in two sections; the first (Section 2.1) of current intermediate representations, their relative strengths and weaknesses, and identification of extensions to a particular intermediate representation; second (Section 2.2), outlines type systems and their abilities to type check programs, and higher order types and their implementation.

Following (Chapter 3) is a research plan, outlining the major steps towards the design and implementation of the proposed extensions to the existing LPVM and Wybe type systems, and the experimental methodology and analyses used to evaluate the effectiveness of these extensions.

Chapter 2

Literature Review

2.1 Intermediate representations

Intermediate representations are abstract languages that exist to aid in the compilation process in a compiler, being an intermediary between source code and compiled machine code. Intermediate representations are designed to better facilitate the transformations and analyses required in compilers to produce more heavily optimised machine code or to provide guarantees about certain properties of a program. The Wybe compiler, `wybemk`, employs a novel Horn clause based intermediate representation, Logic Programming Virtual Machine (LPVM) [5].

2.1.1 Imperative intermediate representations, and SSA and allied forms

The intermediate representations inside the compilation process for imperative languages typically are a derivative or extension of a static single assignment (SSA) form intermediate representation. SSA form was developed by Rosen, et al. [7], and later popularised after an efficient algorithm was devised by Cytron, et al. [8] to transform an intermediate representation into SSA form efficiently. SSA form intermediate representations typically represent the control flow graph as a graph of basic blocks, with each block terminated by a branch to some other block.

An intermediate representation is said to be in SSA form if each variable in a given scope is assigned exactly once. This property allows certain key analyses to be performed more efficiently. For instance, as a variable is assigned once, analyses on the lifetime of the variable are greatly simplified.

Where a variable would be reassigned in a non-SSA form intermediate representation, SSA form introduces a fresh variable which acts as the new version of the variable in later uses of the variable in the control flow graph. These fresh variables propagate naturally through a linear path in a control flow graph. However, as a block can have more than one predecessor blocks, multiple versions of a variable may be available to a block. At the start of one of these blocks, SSA form introduces a ϕ-node to disambigaute which variable is to be used. ϕ-nodes assign the correct version of a variable to a fresh variable within a block. In translation to machine code, these ϕ-nodes are translated to either a move instruction or omitted if register allocation can allocate the variables to the same register. Considerable effort has been made to produce SSA forms with a minimal amount of ϕ-nodes [9, 10], as these nodes do provide some bloat within the intermediate representation, however their utility lies in annotating which blocks provide the values of variables in a given block.

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

FIGURE 2.1: A GCD program in C [5].

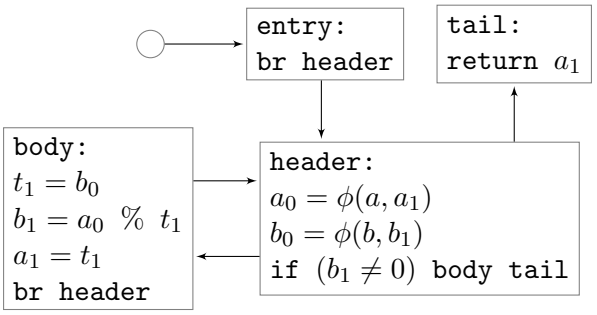

FIGURE 2.2: The GCD program in SSA form [5].

In Figure 2.2, the `header` block has two possible predecessor blocks, `entry` and `body`. To disambiguate which version of a and b are used within the block, the two ϕ-nodes are introduced. If the previous block in execution was `entry`, then the value of a_0 will be a, and b_0 will be b, otherwise they will be a_1 and b_1, respectively.

However, SSA form does come at a cost. ϕ-nodes bias the representation to be amenable primarily to forward analyses. This is due to the ϕ-nodes having no reversed dual in SSA form to indicate which branches use each variable after a branch. This effectively

renders backwards value analyses infeasible, or at the very least offers much weaker analysis, as different branches may assign a different abstract value to the variable.

This problem of forward bias of ϕ-nodes was addressed with the addition of σ-nodes [11] to create static single information (SSI) form. In contrast to ϕ-nodes, σ-nodes are introduced at the end of each branching block and indicate where each variable's values are used next. This allows reasoning in both directions, eliminating the forward bias of ϕ-nodes alone.

Even with the addition of σ-nodes in SSI form, there still exists limitations of value analyses in both SSA and SSI forms. In non-relational analysis, abstract values can propagate through ϕ-nodes by taking the join of each abstract value of the input values. However, for relational analysis, ϕ-nodes cannot propagate information about the relationships between two variables. To remedy this, ϕ-nodes for each variable would have to be refactored to consider the values of all other variables to be able to provide the level of analysis required for relational analysis.

Further additions have been made to SSA form intermediate representations. Gated single assignment (GSA) form [12] introduces γ-nodes. Like ϕ-nodes, γ-nodes are intended to merge different versions of variable at some merge point in the control flow graph. However, γ-nodes additionally require a predicate argument, which defines which branch of the program the executed to reach the merge point. Additionally, thinned gated single assignment (TGSA) [13] defined both μ- and η-nodes, which represent loop headers and loop exits, respectively. In SSA form, ϕ-nodes are used in place of both γ- and μ-nodes, with η-nodes having no equivalent counterpart. These additional nodes allow for reasoning about which predecessor block information flows from, allowing for analyses that include constraints based on the information contained within these extended nodes.

GSA and TGSA, however, do not introduce mechanisms to allow for backwards analyses as was possible in forms such as SSI. The additional nodes also further complicate the intermediate representation.

2.1.2 Functional intermediate representations and continuation passing style

In contrast to SSA form intermediate representations, compilers for functional languages follow a declarative approach. Continuation passing style (CPS) was first used to encode control flow by Strachey [14], however was first used as an intermediate representation in a compiler for Scheme [15]. CPS naturally represents constructs from functional programming naturally, such as closures.

In CPS, control flows via continuations, additional parameters that replace the `return` instruction with a function that uses the returned value. For instance, in Figure 2.3, the parameter `k` is a continuation, and is used in place of the `return` statement, being called with the appropriate return value.

```
gcd(a, b, k) =
    if (b == 0)
    then k(a)
    else let b' = a % b in gcd(b, b', k)
```

FIGURE 2.3: The GCD program in CPS style.

In contrast to SSA-based forms, CPS eliminates name management via different versions of variables by passing parameters to other functions that serve as the basic blocks of the intermediate representation.

SSA form has been shown to be equivalent to a subset of CPS which excludes non-local control flow [16, 17]. Non-local control flow is not used in regular usage of CPS, nor introduced by conventional program transformations, and as such, the two forms can be considered equivalent. Due to this, CPS also suffers from the forward bias that is present in SSA form intermediate representations.

2.1.3 Logic programming form and logic programming virtual machine

As introduced by Gange, et al. [5], logic programming (LP) form intermediate representations utilise Horn clauses as an intermediate representation. Horn clauses [18] are used to represent programs as a series of logical clauses (rules) which are used to make logical deductions to perform some computation. LP form languages are considerably less

complex than SSA and allied forms. Gange also introduces logic programming virtual machine (LPVM) as an implementation of an LP form intermediate representation.

LP form represents a procedure as a goal and a collection of clauses. For a given goal, the collection of clauses are such that, for any given input, exactly one clause can succeed. Clauses also handle name management similarly to how CPS passes variables to other "blocks" via parameter passing. This eliminates ϕ-nodes form the intermediate representation, as variables are either parameters or defined in the clause. Return statements are also not present, as parameters are used to pass data out of procedures, as well as in.

Branches are replaced in the representation via guards. The clauses of a given procedure, pairwise, are identical up to some Boolean guard. After this guard they "fork", diverging from the guard on. Guards are constructed such that all clauses are mutually exclusive and complete, providing the property that for a given input a single clause will succeed. This tames the non-determinism seen in typical uses of Horn clauses (as in Prolog or Mercury), allowing for deterministic evaluation. Unconditional branches are replaced with procedure calls, and loops are replaced with recursive calls.

$$\begin{aligned}&\texttt{gcd}(\texttt{a},\texttt{b},?\texttt{r}) = \texttt{b} \neq \texttt{0} \ \wedge\ \texttt{mod}(\texttt{a},\texttt{b},?\texttt{b}') \ \wedge\ \texttt{gcd}(\texttt{b},\texttt{b}',?\texttt{r})\\ &\texttt{gcd}(\texttt{a},\texttt{b},?\texttt{r}) = \texttt{b} = \texttt{0} \ \wedge\ ?\texttt{r} = \texttt{a}\end{aligned}$$

FIGURE 2.4: The GCD program in LPVM [5].

In a naïve representation of clauses, the common sections of clauses (that which appears before some guard) are replicated. This introduces the possibility of repeating the same analyses on these common parts. Instead, representing clauses as a tree, with guards acting as branch points allows for a more compact representation, that can be used to ensure that transformations keep clauses complete and consistent without repeating any analyses.

$$\texttt{gcd}(\texttt{a},\texttt{b},?\texttt{r}) = \begin{cases} \texttt{b} \neq \texttt{0} \ \wedge\ \texttt{mod}(\texttt{a},\texttt{b},?\texttt{b}') \ \wedge\ \texttt{gcd}(\texttt{b},\texttt{b}',?\texttt{r}) \\ \texttt{b} = \texttt{0} \ \wedge\ ?\texttt{r} = \texttt{a} \end{cases}$$

FIGURE 2.5: The GCD program in LPVM with clauses represented as a tree.

LP form addresses the primary issues with existing intermediate representations while remaining relatively simple in comparison. As standard with logic programming languages, clauses are logically an unordered conjunction, forward and backwards analyses

are both natural in the representation.

As ϕ-nodes and derivatives are not present in the representation, relational analyses can be performed without the difficulties in the presence of such nodes. Similarly, as with a functional intermediate representation, such as CSP, name management is not a concern. Scopes are also explicit in the representation.

However, LP form, and hence LPVM, do lack some features present in other intermediate representations. Higher order functions are not present in LP form intermediate representations. This is contrary to being commonplace in functional languages and their intermediate representations, along with SSA form intermediate representations, such as LLVM [19].

2.2 Type systems and higher order types

2.2.1 Type systems

A type system is a set of rules that define how objects in a language are typed and how these objects can legally be combined to create well-typed programs, aiding the programmer in writing well-formed programs.

Primitive type systems, such as the simply typed lambda calculus ($\lambda^{\rightarrow}$) [20], provide a basis upon which more rich type systems are derived. The types described by $\lambda^{\rightarrow}$ are described in Figure 2.6, with terms t and types T.

$$t ::= x \mid \lambda x : T \;.\; t \mid t\ t$$
$$T ::= C \mid T \rightarrow T$$

FIGURE 2.6: The grammar of $\lambda^{\rightarrow}$.

To check that a program is legally typed, or type checks, a series of rules are used to check that each term is correctly typed in a given typing context, Γ. We say that t has type T if $x : T \in \Gamma$ or $\Gamma \vdash t : T$. The typing rules of $\lambda^{\rightarrow}$ are outlined in 2.7.

Under $\lambda^{\rightarrow}$, T-VAR describes typing of variables, T-ABS describes how λ abstractions (functions) are typed, and T-APP describes how function applications (function calls)

$$\text{T-Var}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\text{T-Abs}\ \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash (\lambda x : T_1\ .\ t) : (T_1 \to T_2)} \qquad \text{T-App}\ \frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 t_2 : T_2}$$

FIGURE 2.7: The typing rules of $\lambda^{\to}$. Each typing rule is read as a natural deduction. Given we can show that the collection of premises above the horizontal line for a given rule are true, we can infer what occurs below the line (the conclusion).

are typed. For instance, if we can infer that some function f has type $Int \to Int$, then the term $f(1 : Int)$ has the type Int

A limitation of $\lambda^{\to}$ is the lack of polymorphic types. An extension to $\lambda^{\to}$, System F, introduces parametric polymorphism to the type system [21, 22].

$$t ::= x \mid \lambda x : T\ .\ t \mid t\ t \mid \Lambda X\ .\ t \mid t\ [T]$$
$$T ::= X \mid T \to T \mid \forall X\ .\ T$$

FIGURE 2.8: The grammar of System F.

$$\text{T-TAbs}\ \frac{\Gamma, X \vdash t : T \qquad \Gamma \vdash t_1 : T_2}{\Gamma \vdash \Lambda X.t : \forall X.T} \qquad \text{T-TApp}\ \frac{\Gamma \vdash t : \forall X.T_1}{\Gamma \vdash t[T_2] : [X \mapsto T_2]T_1}$$

FIGURE 2.9: Typing rules in System F, in addition to those in $\lambda^{\to}$.

T-TAbs allows for a term's type to be abstracted, producing a type variable, X, which represents the abstract type, allowing for the variable type to be instantiated later. T-TApp is the dual of this, applying a type, T_2, to some abstracted type, replacing the instances of the variable type with T_2.

A type system, the Hindley-Milner (HM) type system [23–25] is a popular restriction to the System F type system. HM is the basis of the type system in the ML family of languages, such as Haskell [3]. Where System F allows for types to be universally quantified at any level, HM allows only for types to be qualified at the prenex position (top-level).

For instance, while a type $(\forall X : X \to X) \to (\forall X : X \to X)$ is legal in System F, this is an illegal type in HM. In HM, the type $\forall X : (X \to X) \to (X \to X)$, is legal, but has the restriction that both Xs must be applied to the same type.

Type inference is a process that can be performed in the presence of some type system. Type inference allows to infer the type of some term. If a type system permits type inference, a programmer can omit type annotations when programming and still gain the benefits of the type system, as inferred types are also checked for correctness.

The simply typed lambda calculus has a determinstic algorithm for type inference. A primary benefit of the restrictions of System F present in the HM type system is the ability to perform type reconstruction. If one erases the type annotations of a program in System F, no algorithm exists that can reconstruct the most general types of each term in the program. With HM, once types are erased (or are not included by the programmer), a determinstic, linear time, algorithm can be used to reconstruct the most general type of each term. This allows a legal program in HM type system to lack type annotations entirely while maintaining the guarantees that type errors cannot occur in execution, a property that is lacking if the System F type system is employed.

Type checking can occur at different stages for a given implementation of a language [26]. If types are checked at compile time, then the language is said to be statically type checked. Alternatively, types can be checked at run time, under which the language is said to be dynamically type checked. Static type checking is said to be "type safe" (or sound), as once a program passes the type checking stage in compilation, the program is guaranteed to be safe from runtime type errors. Dynamic type checking does not allow for such guarantees.

However, for a static type system of a Turing complete language to be sound and decidable, that is, to only allow legally typed programs, there must exist legally typed programs that do not pass the type checking stage (it is incomplete). This is due to Gödel's first incompleteness theorem [27], which states that any consistent system that allows for a certain level of elementary arithmetic to be performed is incomplete. Dynamic type checking can overcome this by not being sound, however this again comes at the cost of runtime type errors.

2.2.2 Higher order types

Higher order functions refer to functions (or procedures), that contain a parameter (or return value) that is a function. Higher order functions are commonplace in functional

languages, such as ML and Haskell [3].

Languages that incorporate functions in this manner are said to have functions as first-class citizens in the type system. The distinction between types that have "first-class citizen" status in a type system and those that are "second-class" was first coined by Strachey [26]. Second-class objects refer to objects that must appear "in person", and cannot be referred to by a variable name; first-class contrasts this with objects in the language that can be represented by a variable. An example of a language that does not support first-class functions is the ALGOL language.

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (a:as) = f a : map f as
```

FIGURE 2.10: The `map` function in the Haskell language.

A canonical example of a higher order function is `map`, shown in Figure 2.10. Here, the variable `f` is a function of type `a -> b`. The function takes an argument of type `a` and returns a value of type `b`. The `map` function is also polymorphic in this implementation, as the type variables `a` and `b` can be instantiated to arbitrary types.

The type systems previously mentioned, $\lambda^{\rightarrow}$, System F, and HM, all support higher order types, which have the form $(T_1 \rightarrow T_2) \rightarrow T_3$. In their representation here, functions are curried. Currying [28, 29] allows for functions to be "partially applied", meaning that all arguments do not have to be supplied at the call site. For instance we can bind `fooMap = map foo` and later apply `fooMap` to lists, having the same effect as though `map foo` was applied instead.

While the type system in Wybe features both polymorphic types and type inference, the type system is lacking higher-order types. This is a notable restriction of the type system and the language as a whole. With concepts in Wybe not seen in other languages, such as an alternative parameter passing methods (resources), Wybe presents a novel space for which higher order types can be implemented.

2.2.3 Implementation of higher order types

Higher order types face some challenges in implementation, especially when implemented as first-class members of the type system. Issues faced in implementation of first-class

functions have been coined the funarg (function argument) problem [30], and appears in two forms, the upwards funarg problem and downwards funarg problem. These issues arise when variables are "free" inside the body of a first-class function and defined outside the function.

The upwards funarg problem arises when first-order functions are passed upwards as a result of some function call. Variables are typically stored on a stack frame for a given function, and once the function returns, references to variables on the stack frame are considered invalid. In contrast, the downwards funarg problem occurs when a function is passed downwards as a higher order argument of some function. If a stack frame is to be reused, as in the case of a tail call, references to variables in the stack frame become invalid again.

These functions can largely be avoided by restricting the implementation of first-class functions by either omitting nested functions (as in the C language [31]), or not supporting functions as return values (as in the ALGOL 60 language [32]). Early functional languages, such as Lisp [33], adopted the approach of dynamic scoping. When called, a function would resolve a variable to the closest definition of such a variable at the site of execution. This differs from a lexically scoped approach, where variables inside a function are resolved when the function is defined.

A solution to the funarg problem is to make use of closures [34]. A first class function is represented as a pair of function pointer and a reference to the set of free variables (those that are not defined in the function's scope).

The environment of a closure, capturing the free variables in the scope of the closure, can have different representations [35]. Environments can be flat, being represented as a flat record or vector containing a copy of each free variable. Alternatively a closure's environment can be represented in a linked structure. In this linked structure, the environment contains references to other environments. These references allow for closures to share environments, for instance if the scope of a closure is contained within another with some shared free variables, the environment of the inner closure can contain a reference to the environment of the outer closure to gain access to those free variables.

Of course, each of these representations have benefits. Flat environments allow for faster retrieval of a free variable, requiring a single fetch instruction. In the case of

linked environments, retrieving a closed variable requires traversing deep into the linked structure until a reference to the free variable is found. In contrast, flat environments do not allow for environments to be reused or shared between closures. This leads to memory overheads associated with flat environments that are not shared with linked environments. With flat environments generally requiring larger contiguous segments of memory, allocation of memory for them may also be slightly slower than that of a linked environment, which generally has a smaller memory footprint.

These benefits, however, may not be clear cut. If closures share the same scope, the flat representation of their environment can be merged into a single record or vector, allowing for some sharing of environments. This can also be performed for linked environments in the same scope if beneficial.

Linked environments also have the potential of using too much space. The linked structure will require some form of garbage collection to maintain. If some deeply linked environment is unused, a garbage collector may not realise this memory can be reclaimed.

Closures are created within the compiler in transformations of its intermediate representation. This process, called closure conversion, allocates space for a closure's environment, marshalling free variables. Closure conversion has been extensively optimised for time and space due to its potential to cause overhead over a regular procedure call [36, 37], and has been shown to be safe for space and time. Various techniques also exist that allow for greater space saving if the function pointer is known at the call site or the number of free variables is small (one or two) [38].

Closure conversion is similar to lambda lifting [39], which hoists nested functions to the global scope, and closes free variables by passing them as arguments to the lifted function. The inverse operation of lambda lifting is lambda dropping [40], which is used to reduce the scope of a function, leading to a reduction in the number of parameters and hence easier analysis.

Intermediate representations also play a role in optimising the implementation of closures. For instance, a graph-based higher order intermediate representation, Thorin [41], allows for certain closures to be eliminated entirely. As this intermediate representation

supports closures natively, a transformation pass has been created that is used to remove certain types of closures from programs entirely. The transformation pass extends the concept of lambda lifting and lambda dropping into a single transformation called lambda mangling, which can remove the use of closures in a wide class of programs which are tail recursive. This is further shown to reduce the overheads associated with closures to be negligible, with comparative performance when compared to first order code.

Chapter 3

Research Plan

The plan for this research is broken into three stages that follow the research questions.

1. Formalise and implement a higher-order extension to the existing LPVM intermediate representation (Section 3.1)
2. Formalise the existing Wybe type system and extend this type system to support higher-order types, then implement this type system in the current Wybe implementation (Section 3.2)
3. Evaluate the performance of higher-order constructs in Wybe (Section 3.3)

3.1 Logic programming virtual machine extension

The intermediate representation that Wybe uses, LPVM, does not have support for higher order types. In order to support higher order types in Wybe, LPVM must be extended to natively support higher order types.

The state of the current type system incorporated with LPVM is undocumented. Developing a set of typing rules and semantics would benefit the state of the LPVM implementation. With this developed typing rules and semantics, extensions, such as the proposed higher order extension, can be founded upon a rigorous type system.

The development of these typing rules, semantics, and the higher order extension will require a series of proofs of the system being sound. This provides the rigorous foundation upon which further extensions can be derived.

Once completed, the implementation of LPVM in the Wybe compiler can also be extended to make use of the higher order extension. This will also require mapping constructs in LPVM to LLVM constructs in order to be used in the Wybe compiler. Further, if holes exist in the existing implementation that can be uncovered with formal reasoning, the implementation can be made more rigorous.

3.2 Wybe extensions

With the Wybe compiler making use of LPVM as its primary internal representation, after extending LPVM to support higher order types, we can implement a higher order type system for Wybe.

Wybe supports numerous features that are unique to, or at least not largely seen, in the language design space. Common with languages such as Prolog, arguments to a procedure call can be in different modes, with arguments as an input or an output, and further in Wybe can be both an input and an output. Features such as the overloading of modes and resources in Wybe provide a point of novelty towards the extended type system of Wybe.

As with LPVM, the type system of Wybe is not formally documented. With the creation of typing rules and corresponding proofs, we can provide a formal basis for the Wybe type system and extensions to the type system.

Higher order types and resources is an untravelled road. With the state of a resource requiring being thread through the programs structure, issues regarding how these resources can be passed arise. Generally, we can consider four approaches towards the design of resourceful higher order procedures, however which option we will implement is currently unknown until we can determine how we can ensure interface integrity with some of the less restrictive options.

One option is to disallow arguments to higher order procedures to use resources. This option is not ideal, as it significantly limits the utility of such procedures as any resourceful procedure cannot be passed as an argument.

A less constrained approach is to allow higher order types to specify the mode and type of resources that will be used. A limitation of this system is the requirement that, even if two or more resources have the same type, the use of each resource in a higher order type would require an overloaded definition of the resourceful procedure. These overloaded definitions could share the same body, yet the source code would require duplication of the body, leading to poor language design.

To contrast the restrictive approach mentioned above, higher order procedures could remove the annotation of which resources each parameter could use yet still allow for the parameters to use resources. Instead, when calling a higher order procedure, if the parameters use resources, the program could transform the call to a specialised form that makes use of the provided resources, akin to the previous approach. This possibly also violates a leading design principle behind Wybe of interface integrity. This is due to the interface of a resourceful procedure not specifying that the parameters could make use of a resource.

Allowing resources used by a procedure to be defined polymorphically is a further approach that could be used as a compromise between the two former approaches, and will be the approach used in this research. Contrasting the second approach of having all resource usage explicit, this approach would allow a resource to be defined to have a polymorphic type. This alleviates the restriction of the previously mentioned approach requiring duplication of procedure's bodies in the source code.

Wybe also allows for procedures to be tagged based on certain properties. The `test` or `partial` tag specifies a function that may fail, for instance, or `terminal` that specifies that a procedure will never return. Ideally, higher order types should also be able to specify if the procedure is tagged, allowing for the compiler to perform certain optimisations with higher order types.

The implementation of higher order types in programming languages is typically performed with the use of closures, and will be the adopted implementation strategy here. The elimination of closures, however, has been shown to have a significant performance

increase [41], allowing for higher order code to perform as efficiently as first order code. As such, wherever possible, closures will be attempted to be eliminated, such as through inlining.

Wybe's syntax is also unique, and supporting extensions to the language will require new syntax for these extended features. In place of currying and partial application as seen in languages such as Haskell and ML, we can perform similar effects through the use of anonymous procedures. We propose a new syntax for anonymous procedures that make use of "holes". In this proposed syntax, a hole (`@`), can be used in place of a variable, meaning that this argument is to be specified later, forming forming the interface of the anonymous function. For holes that are in output mode, the prefix ? is used, as is standard in Wybe currently.

For example, `?fooBar = { foo(@, bar, ?@) }`, binds to `fooBar` an anonymous procedure that calls `foo`, with the first argument as the first input of the anonymous procedure, the second with the value of `bar`, and the third as the output of the anonymous procedure. As `bar` is not defined inside the body of the anonymous procedure, this anonymous procedure must be a closure with `bar` as a free variable.

To extend this syntax to be more flexible, the position of an argument can be specified with a positive integer after each `@`. For example, a call to `map` with an anonymous procedure could be represented as `?out = map({?t = @1 + sin(@1); ?@2 = t * cos(t)}, in)`, which binds `out` to the list which contains the anonymous procedure (between the braces) applied to each element of `in`. This also allows for the same input to be used multiple times in the anonymous procedure, which is not possible succinctly without the numbering scheme.

With this numbering scheme, we can also reorder the arguments of an anonymous procedure. For instance, `?fooBar = { foo(@2, bar, ?@1) }` would bind `fooBar` as above, however with the closure's arguments in reversed order.

3.3 Evaluation

In order to evaluate the implementation of higher order programming in Wybe, two factors will be considered: program execution time and program executable size.

Execution is a widely used metric for evaluating the effectiveness of the implementation of a language feature and the performance of such a feature [41–43]. As is typical in the testing of performance via execution time, each test will be executed numerous times, with the average time taken as the "true" time. Averaging in this way allows for a control over factors that may effect the runtime of a program due to varying loads on a CPU in varying runs of each test.

Executable size will also be evaluated to ensure that the extensions to Wybe do not cause a significant increase in the executable size of Wybe programs. While it is not used widely as a metric to evaluate the performance of a language feature, this property is worth evaluation as there is potential for exponential blow-up in the size of executables if the implementation of higher order constructs is not carefully considered

Currently, there does not exist a large sample of non-trivial Wybe programs. To evaluate the performance of Wybe code, various benchmark programs will be synthesised (sourcing from the Computer Language Benchmarking Game [44], a widely used benchmarking suite [41, 42]) or borrowed from existing sources (such as the N-body problem in Wybe [42]). These programs will incorporate, or be modified to include, higher order code constructs.

To effectively evaluate the performance of higher order Wybe programs, we require a baseline. To act as such a baseline, the set of programs previously mentioned will be translated to into first order code (by inlining and flattening higher order code).

For higher order code to be useful in the Wybe language, the performance of such code should be close to that of equivalent first order code. If it is found that there is a significant bottleneck in the performance of Wybe code in the presence of higher order constructs, it is likely that a redesign or different implementation of the compiler would be necessary. If it is found that the size of executables is much larger in the presence of higher order code, then this also indicates the implementation may be poor.

The hypothesis for the evaluation of execution time, however, is that there will be an increase in execution time, corresponding to a decrease in performance when using higher order code. This hypothesis is due to the overheads associated with higher-order constructs, such as closure construction and calling.

While an increase in execution time seems like a serious drawback to using higher order constructs, the increased expressiveness bought by higher order types adds richness to the language. With a small increase in the execution time, the increased benefit of increased expressiveness may be beneficial, and outweigh the slight increase in execution time. Ideally, this performance overhead should aim to be as small as possible.

Further, the hypothesis is that higher order code will have multiple effects on the executable code size of Wybe programs. The creation and execution of closures will cause an increase in the amount of code generated, as closure creation and calling requires extra instructions. However, the functional paradigm shift can allow for greater reuse of code, especially in tandem with generic types, leading to decreases in other aspects of an executable's size, such as a reduction of duplicated code.

3.4 Timeline

The general timeline for the proposed research is outlined in the Gantt chart in Figure 3.1. As the Wybe compiler makes heavy use of LPVM to perform various analyses and transformations, it is pertinent that the LPVM implementation occurs before the Wybe implementation.

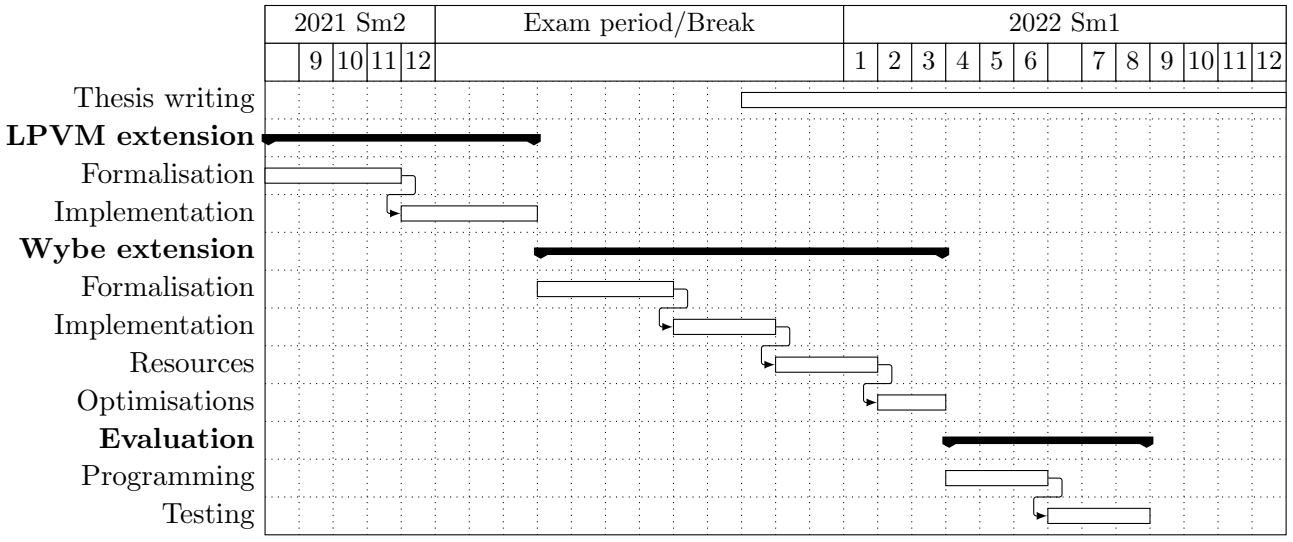

FIGURE 3.1: Proposed research timeline.

Bibliography

[1] Peter Schachte. Wybe: A programming language supporting most of both declarative and imperative programming, 2015. URL https://github.com/pschachte/wybe.

[2] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *History of programming languages—II*, pages 331–367. 1996.

[3] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[4] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Understanding the use of lambda expressions in java. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–31, 2017.

[5] Graeme Gange, Jorge A Navas, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. Horn clauses as an intermediate representation for program analysis and transformation. *Theory and Practice of Logic Programming*, 15(4-5):526–542, 2015.

[6] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming*, 29(1):17–64, 1996. ISSN 0743-1066. doi: https://doi.org/10.1016/S0743-1066(96)00068-4. URL https://www.sciencedirect.com/science/article/pii/S0743106696000684. High-Performance Implementations of Logic Programming Systems.

[7] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 12–27, New York,

NY, USA, 1988. Association for Computing Machinery. ISBN 0897912527. doi: 10.1145/73560.73562. URL https://doi.org/10.1145/73560.73562.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, 13:451–490, 1991.

[9] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4(1-10):1–8, 2001.

[10] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. Simple and efficient construction of static single assignment form. In Ranjit Jhala and Koen De Bosschere, editors, *Compiler Construction*, pages 102–122, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37051-9.

[11] C. Ananian. The static single information form. Master's thesis, Princeton University, 2001.

[12] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, page 257–271, New York, NY, USA, 1990. Association for Computing Machinery. ISBN 0897913647. doi: 10.1145/93542.93578. URL https://doi.org/10.1145/93542.93578.

[13] Paul Havlak. Construction of thinned gated single-assignment form. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 477–499. Springer, 1993.

[14] Christopher Strachey and Christopher P Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-order and symbolic computation*, 13 (1):135–152, 2000.

[15] Guy L Steele Jr. Rabbit: A compiler for scheme, 1978.

[16] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. *SIGPLAN Not.*, 30(3):13–22, March 1995. ISSN 0362-1340. doi: 10.1145/202530.202532. URL https://doi.org/10.1145/202530.202532.

[17] Andrew W. Appel. Ssa is functional programming. *SIGPLAN Not.*, 33(4):17–20, April 1998. ISSN 0362-1340. doi: 10.1145/278283.278285. URL https://doi.org/10.1145/278283.278285.

[18] Alfred Horn. On sentences which are true of direct unions of algebras1. *The Journal of Symbolic Logic*, 16(1):14–21, 1951.

[19] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis amp; transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004. doi: 10.1109/CGO.2004.1281665.

[20] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940. doi: 10.2307/2266170.

[21] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Éditeur inconnu, 1972.

[22] John C Reynolds. Towards a theory of type structure. In *Programming Symposium*, pages 408–425. Springer, 1974.

[23] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[24] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[25] Luis Damas. Type assignment in programming languages. *KB thesis scanning project 2015*, 1984.

[26] Christopher Strachey. Fundamental concepts in programming languages. *Higher-order and symbolic computation*, 13(1):11–49, 2000.

[27] Kurt Gödel. *On formally undecidable propositions of Principia Mathematica and related systems*. Courier Corporation, 1992.

[28] Gottlob Frege. *Grundgesetze der Arithmetik: begriffsschriftlich abgeleitet*, volume 1. H. Pohle, 1893.

[29] Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin. *Combinatory logic*, volume 1. North-Holland Amsterdam, 1958.

[30] Joel Moses. The function of function in lisp or why the funarg problem should be called the environment problem. *ACM Sigsam Bulletin*, (15):13–27, 1970.

[31] Dennis M Ritchie, Brian W Kernighan, and Michael E Lesk. *The C programming language*. Prentice Hall Englewood Cliffs, 1988.

[32] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger, and Peter Naur. Report on the algorithmic language algol 60. *Commun. ACM*, 3(5):299–314, May 1960. ISSN 0001-0782. doi: 10.1145/367236.367262. URL https://doi.org/10.1145/367236.367262.

[33] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[34] Erik Sandewall. A proposed solution to the funarg problem. *SIGSAM Bull.*, (17):29–42, January 1971. ISSN 0163-5824. doi: 10.1145/1093420.1093422. URL https://doi.org/10.1145/1093420.1093422.

[35] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 293–302, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897912942. doi: 10.1145/75277.75303. URL https://doi.org/10.1145/75277.75303.

[36] Zhong Shao and Andrew W. Appel. Efficient and safe-for-space closure conversion. *ACM Trans. Program. Lang. Syst.*, 22(1):129–161, January 2000. ISSN 0164-0925. doi: 10.1145/345099.345125. URL https://doi.org/10.1145/345099.345125.

[37] Zoe Paraskevopoulou and Andrew W. Appel. Closure conversion is safe for space. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi: 10.1145/3341687. URL https://doi.org/10.1145/3341687.

[38] Andrew W. Keep, Alex Hearn, and R. Kent Dybvig. Optimizing closures in o(0) time. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*, Scheme '12, page 30–35, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450318952. doi: 10.1145/2661103.2661106. URL https://doi.org/10.1145/2661103.2661106.

[39] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. pages 190–203. Springer-Verlag, 1985.

[40] Olivier Danvy and Ulrik P Schultz. Lambda-dropping: transforming recursive equations into programs with block structure. In *Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 90–106, 1997.

[41] Roland Leißa, Marcel Köster, and Sebastian Hack. A graph-based higher-order intermediate representation. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 202–212, 2015. doi: 10.1109/CGO.2015.7054200.

[42] Zijun Chen. Multiple specialization for the wybe programming language. Master's thesis, The University of Melbourne - School of Computing and Information Systems, 2020.

[43] Shijie Xu, David Bremner, and Daniel Heidinga. Fusing method handle graphs for efficient dynamic jvm language implementations. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL 2017, page 18–27, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450355193. doi: 10.1145/3141871.3141874. URL https://doi.org/10.1145/3141871.3141874.

[44] Isaac Guoy. The computer language benchmarks game, 2000. URL https://benchmarksgame-team.pages.debian.net/benchmarksgame/.