Higher-Order Programming in Wybe

by

James Barnes
Student Number: 820946

Supervised by Dr. Peter Schachte

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

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

June 2022

The lurking suspicion that something could be simplified is the world's richest source of rewarding challenges.

— Edsger *Wybe* Dijkstra

THE UNIVERSITY OF MELBOURNE

Abstract

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

Master of Computer Science

by James Barnes
Student Number: 820946

Higher-order programming is a paradigm marked by increased expressiveness and greater modularisation. With higher-order programming, functions and procedures are promoted to first-class citizens of the language, allowing for such increased expressiveness. In this thesis, we introduce a higher-order extension to the Wybe language.

Wybe is a multi-paradigm language that exists as both an imperative and declarative language, and features the strong, static, guarantees of many declarative languages, while featuring many imperative constructs. Features that are common in logic programming languages, such as a mode system, are also present in the Wybe language. One notable feature of the language is the resource system. Resources are akin to global variables of imperative languages, however, their usage is restricted to attain declarative properties. Resources have not been explored yet in a higher-order context, and with the semantics we explore for higher-order resources, we introduce a novel implementation strategy that makes use of global variables.

Along with higher-order terms, global variables are also required in the intermediate representation used within the Wybe compiler, LPVM. We extend LPVM to support higher-order terms and global variables. Further, we extend the current optimisations of LPVM and introduce novel optimisations to reduce the manipulation of global variables where possible.

We also investigate the effects of the extensions to the language and intermediate representation. We compare the execution times and program size of the current and extended resource implementation, and the use of first-order and higher-order code. We show that the performance is generally comparable to the performance of the current implementation of the language, with speed-ups and slowdowns in certain cases.

Declaration of Authorship

I, James Barnes, declare that this thesis and the work presented in it are my own. I confirm that:

- this thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text.
- this thesis did not require clearance from the University's ethics committee.
- the thesis is approximately ./scripts/fmt-texcount.sh: line 3: texcount: command not found words in length (excluding text in figures, tables, code listings, bibliographies, and appendices).

Signed: 

Date: June 2022

Preface

This thesis focuses on an established language, Wybe, that existed before the commencement of this piece of work. All code written by myself in support of this piece of work was independently reviewed by my supervisor, Dr. Peter Schachte. Feedback received was then further incorporated into the code. Any previously existing code is documented as being such throughout this document. Feedback was also received from my supervisor throughout the creation of this thesis and was incorporated into the final product.

The literature review portion of this thesis is taken from, and further modified from, the research proposal that was undertaken to fulfill another portion of the requirements of the Master of Computer Science. This literature review was ultimately inspired by the work of myself in the completion of COMP90044 (Research Methods), prior.

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 on and reside in. I pay my respects to the elders, past, present, and future.

I would also like to acknowledge the assistance lent to me by my supervisor, Dr. Peter Schachte. His continued guidance of me throughout this learning journey is greatly appreciated, and his wealth of knowledge and insight has been priceless. The time lent to me to talk through problems faced throughout the production of this work is greatly appreciated.

Finally, I would like to acknowledge the support I have received from my family, partner, and friends throughout this degree. Their support through this process has been a wealth of comfort.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Wybe (pronounced ['wi.bə], WEE-BUH) is a programming language created at the University of Melbourne by Dr. Peter Schachte [1]. Wybe draws inspiration from both logic programming and imperative programming paradigms, taking concepts such as modes and determinism from logic programming into a language with a programming style of imperative languages.

1.1 Problem Statement

Currently, the Wybe language supports only first-order programming, where all terms have first-order types. In this thesis, we introduce an extension to the Wybe language that supports higher-order programming.

While many languages support higher-order programming, the Wybe language supports features that, to our knowledge, have not been investigated in a higher-order context. One such novel language feature Wybe has is a resource system. Resources in Wybe are akin to global variables in imperative languages or class variables in object-oriented languages. Resources, however, are more constrained in their usage, with each procedure that manipulates a resource requiring being marked as such.

The resource system has not been investigated in the context of higher-order programming, providing a space in which we can investigate this language feature in a higher-order context. The desired semantics motivate an extension to the intermediate representation that introduces global variables. We aim to extend the Wybe language with

our desired semantics of resources and provide novel optimisations enabled by these semantics.

With these extensions to the Wybe language, we evaluate the performance of the language in terms of execution time and program size, with the existing language as a baseline. While a slow-down may seem detrimental to the utility of a language, higher-order programming increases the expressiveness of a language. The increased expressiveness allows more general programs to be written with less source code and programming effort. Ideally, these overheads should be relatively small in comparison to the overall runtime of a program, lowering the cost of these extensions.

Hence, we aim to answer these research questions:

1. How can the Wybe language be extended with higher-order programming?
2. How can the resource system of Wybe be extended to support higher-order programming while maintaining the guarantees of resources and the expressiveness of higher-order programming?
3. Do these extensions perform similarly in terms of execution runtime and program size when compared with the existing Wybe language?

1.2 Document Outline

The remainder of this chapter continues with an overview of the Wybe language. Following, in Chapter 2, we review literature regarding the design an implementation of higher-order programming, type systems, and intermediate representations.

As the Wybe language currently does not support higher-order programming, we introduce new syntactic structures to support higher-order terms in Chapter 3. We add functionality to handle common features present with higher-order types, such as partial applications and closures. We define the syntax and semantics of these features, along with the semantics we define for resources in a higher-order context.

In support of a formal basis for the Wybe type system, in Chapter 4, we define a formalised form of the Wybe type system, with declarative semantics. This formalised system provides the basis for which the Wybe type checking algorithm is born. We also

outline the heuristics in place for the compiler's type checking algorithm and discuss the higher-order extensions to this algorithm.

In Chapter 5, we devise an implementation strategy that promotes resources into global variables, motivated by the semantics of resources in a higher-order context. We discuss the existing translation of resources into formal parameters, and also discuss a limitation of globalised resources.

Following, in Chapter 6, we outline the translation process of the Wybe AST into LPVM, the intermediate representation used within the Wybe compiler, and extend this translation to transform higher-order terms into closures. We also extend LPVM, accordingly to support global variables, and introduce *global flows*, a declarative interface that defines the flow of information of global variables.

In Chapter 7, following the extension of LPVM, we extend optimisations currently used within the Wybe compiler to support the extensions with higher-order programming and global variables. We also introduce a novel optimisation that constrains the manipulation of global variables with the aid of global flows.

In Chapter 8, we discuss the translation of the extensions of LPVM into LLVM, the final intermediate representation used by the Wybe compiler.

Finally, in Chapter 9, we investigate the effects of the extended language on various program metrics. This includes the usage of higher-order programming and the implementation of resources using global variables, and the effects on the runtime of programs. We compare the runtime across several programs to gain insight into the performance of the language in the presence of higher-order programming and global variables. We also investigate the size of Wybe programs regarding the usage of higher-order programming and the use of global variables.

1.3 The Wybe Language

This section is intended to give the reader an overview of the Wybe language. This should not be considered complete documentation for the language or a complete introduction. It should, however, give the reader an understanding of the syntax and semantics of the

Wybe language, to understand the Wybe code fragments that appear throughout the body of this thesis.

The Wybe language's defining principle is a concept called *interface integrity*. Interface integrity is the property that, through the interface of a procedure, all information that flows into and out of a procedure should be known. This ensures that there are no hidden effects of a call — you can always tell what effects a call could have without needing to look at its implementation. Such effects can be seen from various procedure modifiers also, such as that of purity, and through the explicit usage of resources.

Wybe employs *copy on write* semantics. Copy on write semantics allows for efficient copying of data by simply copying a reference to the data, while modifications may be slower. In general, data is not copied until it is rewritten, in which case the original data is copied and then is destructively overwritten. As such is possible in Wybe for multiple variables to alias the same data, and to save time copying data, we delay copying data until it is necessary by instead sharing references to the same data when aliased. Only data that is definitely un-aliased can be destructively modified, otherwise, the data must first be copied before the new copy can be destructively modified.

1.3.1 Modules

A Wybe program is divided into discrete sets of items called modules. A module loosely corresponds to a source code file. The file system's structure dictates the module structure between various source files.

Each module contains a series of items, with each having an optional publicity modifier (`pub`) to indicate this item is visible outside this module. All items are implicitly visible to all submodules. These items are broadly divided into the following categories:

- Imports
- Top-level code
- Submodule declarations
- Type constructors or type representation
- Procedure and function declarations

- Resource declarations

Imports

Import statements allow for external modules to be loaded. An import statement (`use`) is followed by a list of modules, each of which has its public items available for use in the current module. Imports can also be used to load libraries or modules from a foreign language, such as C, allowing the use of foreign code from some other language.

Top-level Code

Top-level code consists of all code that is present at the top-level of a Wybe module. This code is used in the creation of the executable file when compiled, combined with top-level code from other modules. For example, the "Hello, World!" program can be written as in Listing 1.1, as the `println` statement exists at the top-level of the program.

```
1 !println("Hello, world!")
```

Listing 1.1: "Hello, World!" in Wybe.

Submodule declarations

Modules can also contain other modules, called submodules. These submodules implicitly have access to the parent module's items, however, the parent module only has access to the submodule's public items.

Type constructors or representation declarations

Each Wybe module can also be declared as a type. Submodules that are declared as being a `type` also allow for such constructor declarations. These constructors are further discussed in Section 1.3.4.

A Wybe module that is a type can either be an algebraic data type or have a low-level type representation. Low-level type representations allow for primitive types, such as

signed or unsigned integers, floating-point numbers, and raw memory addresses to be used within the Wybe language.

Procedure and function declarations

In Wybe, there are two syntactically distinct constructs used to define blocks of code. These are procedures and functions. Procedures allow for the definition of a named block of code, the body, with appropriate inputs and outputs (preceded by a `?`). Functions are similar, yet have an implicit output value that is the expression that is the body of the function.

```
1 def proc(i:int, j:int, ?k:int) {
2     ?k = i + j
3 }
4
5 def func(i:int, j:int):int = i + j
```

Listing 1.2: Procedure and function declarations.

In Listing 1.2, we define a procedure, `proc`, with an equivalent function definition, `func`. All function definitions are syntactic sugar for procedure definitions in Wybe, and as such, we will focus on procedures throughout the remainder of this thesis.

Resource declarations

Resource declarations allow for a resource to be defined with a given type, with an optional initialisation value. A resource is the replacement of global variables in Wybe and is further detailed in Section 1.3.5.

1.3.2 Modes

Wybe features a strong mode system. The mode system defines which arguments or parameters are treated as inputs, outputs, or both. Each procedure or function declaration defines an interface consisting of a series of parameters, with each parameter prefixed by a flow specifier, which defines the direction data flows via this parameter. A parameter

adorned by a preceding `?` denotes that this parameter is an output to the procedure. For parameters with no prefix, the parameter is considered an input.

A parameter preceded by a `!` is considered both an input and output. The parameter begins with a value, and can then be modified throughout the body of the procedure, with the final value being passed as an output from the procedure.

In general, a procedure may be defined with any argument in any mode. This means that the number of outputs of a procedure is also unrestricted, unlike in most conventional programming languages.

Multiple procedures can be mode overloaded versions of the same procedure. This allows a Wybe program to define procedures "in reverse", similar to how languages such as Prolog and Mercury offer multiple modes. However, Wybe requires *different* procedure bodies for each mode, unlike in Mercury and Prolog where each procedure may be used in different modes but with the same definition by default.

```
1 def add( x:int,  y:int, ?z:int) { ?z = x + y }
2 def add( x:int, ?y:int,  z:int) { ?y = z - x }
3 def add(?x:int,  y:int,  z:int) { ?x = z - y }
4 def add(!x:int,  y:int)         { ?x = x + y }
5
6 add( 1, ?x, 2) # x = 1
7 add(?x,  3, 2) # x = -1
8
9 add(!x, 3) # x = 2
```

Listing 1.3: Procedure declarations with different arities and modes.

In Listing 1.3 we define the procedure `add` with two different arities and 4 different modes. The first is used where the first and second parameters are inputs and the final is an output parameter; the second is used where the second parameter is the output; the third where the first parameter is an output. On line 6 we use the second defined mode of `add`, and on line 7 we use the third defined mode of `add`.

The final declaration differs in arity, and has the first argument as an input and an output, and the second as an input. The final call to `add` on line 9 uses the previously assigned value of `x`, and re-assigns `x` to `x + 3`.

1.3.3 Control Flow

In Wybe, there are three basic forms of control flow: procedure or function calls, branching, and loops.

Procedure and function calls are the simplest control flow mechanism. A procedure or function can be called by providing the (possibly module qualified) procedure name, followed by a list of arguments with their modes adorned. There may be multiple procedures that match the name (or module) of the call, and which procedure is intended to be called is attempted to be resolved during the type checking process. A procedure or function call binds variables that are outputs of the procedure, with outputs marked with a preceding `?` as in procedure declarations. Similarly, arguments marked by a preceding `!` are an input and an output, being reassigned accordingly.

Branching is a simple control flow mechanism that allows computation to branch based on some conditional statement. In general, there may be numerous branches, each with a condition, however, within this thesis, we allow for a branching statement to have two branches, with the second branch being executed if the condition fails to hold. An example conditional branch can be seen in Listing 1.4.

```
1 ?x = 3
2 if { x % 2 = 0 ::
3     !println("x was even")
4 | else ::
5     !println("x was odd")
6 }
```

Listing 1.4: Example conditional branches in Wybe.

Branches bind variables that are bound in all (non-terminal) branches. *I.e.*, if a variable is bound in the condition or the first branch, and it is bound in the second branch, then the variable is bound after the branch.

Loops are the final control structure that we introduce, as seen in Listing 1.5. A `do` block represents such a loop, a list of instructions that are executed repeatedly. There are two special instructions, `next` and `break`, that, respectively, allow for control over the loop structure. The `next` instruction causes the loop to execute again from the start.

The `break` instruction causes the loop to exit, resuming execution from after the loop body.

```
1  ?bottles = 99
2  do {
3      if { bottles = 1 ::
4          !println("1 bottle of beer on the wall, 1 bottle of beer.")
5          !print("Take one down and pass it around, ")
6          !println("no bottles of beer on the wall")
7          break
8      | else ::
9          !print(bottles)
10         !print(" bottles of beer on the wall, ")
11         !print(bottles)
12         !println(" bottles of beer.")
13         !print("Take one down and pass it around, ")
14         ?bottles = bottles - 1
15         !print(bottles)
16         !println(" bottles of beer on the wall.")
17     }
18 }
```

Listing 1.5: Example loop in Wybe, printing the "Bottles of Beer" song. [2]

As a loop is never guaranteed to execute all instructions, a loop will only bind variables that are bound at the end of the loop's body and upon each unconditional `break`.

1.3.4 Types

In Wybe, all types are modules, but not all modules are types. Types that are modules have two forms: algebraic data types and low-level representations. The Wybe compiler automatically generates procedures that construct and deconstruct algebraic types into low-level representations which are used in the final stages of code generation in LLVM.

A Wybe module that is an algebraic data type is marked by a `constructor` declaration, followed by a series of `|`-separated constructors. Each constructor consists of a name and a (possibly empty) list of fields, each of which has a type and an optional name. Constructors are much like that in a language such as Haskell, lacking any source-defined body, contrasting constructors in a language such as Java where the constructor is defined in the source code. Named fields implicitly define a getter and setter procedure,

which are used to get the value of the field and to construct a new value with the same values up to the field that is set, respectively.

For example, in Listing 1.6, the `number` type implicitly defines constructors and deconstructors for the `natural` and `rational` constructors, and a (`test`) procedure to get the `nat` or `rat` field from a respective value of type `number`, with these procedures failing if the wrong constructor was used.

```
1 type number { natural(nat:int) | rational(rat:float) }
2 type tree(X) { nil | node(left:tree(X), X, right:tree(X)) }
```

LISTING 1.6: Example abstract data types in Wybe.

Wybe also supports generic programming. Type variables are declared in the source code as a single uppercase letter followed by zero or more digits (`[A-Z][0-9]*`) (*e.g.*, `X` in Listing 1.6). Algebraic types can also be defined polymorphically, being defined with some number of type parameters, which can then be instantiated to some other type when the type is used. An example of such a polymorphic type is the `tree` type in Listing 1.6, which is defined recursively.

As Wybe uses a static type system, the types of all variables are checked for correctness at compile time. This ensures that a type error cannot occur at runtime. Wybe also features type inference. This allows for the types of variables in a program to be inferred, permitting correctly typed programs to be compiled while lacking type annotations.

One of the distinctive features of the Wybe type system is that of how types can be inferred. The property that is distinct in the Wybe type systems is that types of procedure parameters can only be inferred from the declarations of a procedure itself, not from the contexts where the procedure is called. This property of the Wybe type system makes it distinct from other type systems where the types of a procedure's parameters can be inferred by the contexts wherein it is called.

1.3.5 Resources

Resources are a novel parameter-passing mechanism featured in the Wybe language. Resources are similar to global variables found in imperative languages, class variables

found in object-oriented languages, or the state monad seen in functional languages. Unlike global variables, however, usage of a resource is explicit, and as such stronger optimisations can be performed than could with a global variable.

The intent of a resource is to store a value which may be used throughout the computation, being passed through many calls, yet there is only one of these values in a given computation. Whereas conventionally arguments are passed positionally, a resource is a parameter that is passed implicitly by name, which allows the programmer to automatically thread a resource's state through calls. The value of a resource can be accessed as if it were a variable, and (re-)assignments of this variable will (re-)assign the resource.

A procedure can be defined to `use` some resource. This is annotated in the interface of each procedure, with each used resource having an accompanying flow. To call a procedure that uses some resource, then the resource must be in scope, and the call to the resourceful procedure must be marked with a preceding `!` to ensure a reader is aware a resource is being used. A resource is in scope if it is called within a procedure that uses the same resource, or in a scoped `use` block.

For a resourceful procedure call, some conditions must be met. For inwards flowing resources, the resource is considered an input to the procedure. This means that the resource must not only be in scope for the call to be valid but also must be bound to a value before the call. The value of the resource after the call will be the same as after the call. Outwards flowing (`?`) resources allow for the procedure to be called where the resource is in scope but not necessarily bound. The resource is then bound after the call. Finally, a resource with an input and output flow (`!`) must not only be in scope but also must be bound before the call. The value of the resource may change after the call. Examples of each flow and correct usage can be found in Listing 1.7.

Resource usage can also be scoped in a `use` block. All resources that are named for the `use` block are in scope for the entirety of the intervening statements. Scoped resources are considered bound if a variable of the same name was bound just before the `use` block, otherwise, if there was no variable bound before the block, the resource is considered unbound. The value of any resource (or variable of the same name) after the block is saved before the block and restored after. If the resource was unbound before the scope, the resource remains unbound after the scope. In Listing 1.8 the `strings` resource is used in a `use` block, introducing a scope wherein the resource can be used.

```
resource res:int

def in(?x:int) use res {  # value of res can be used
    ?x = res
    ?res = res + 1
}
def out(x:int) use ?res { # value of res can only be used once set
    x = ?res
}
def inout use !res {      # value of res can be used, and modified
    ?res = res = 1
}

use res in {
    ?res = 1  # assign res a value
    !in(?x)   # res is implicitly passed in, also remains unchanged
    !out(x)   # res is implicitly passed out, value before is irrelevant
    !inout(x) # res is implicitly passed in/out
}
```

LISTING 1.7: Example resource flows

In the standard library of the Wybe language, a resource `io` is defined. The `io` resource is designed to provide a pure interface via which declarative I/O is performed, with the resource representing the state of the entire world. Each I/O procedure is defined with `use !io` in their interface, which ensures that the state is fed through the entire computation where I/O is performed. The `io` resource has a *phantom* type, being a 0-bit integer type, that is used throughout compilation to ensure the ordering of I/O operations while containing no information, and as such is eliminated in the late stages of code generation. This provides declarative I/O with zero overhead.

In Listing 1.8 we see an example of resource usage. The resource, `strings`, is declared with the type of `list(string)`. We define two procedures that `use` the `strings` resource in the in/out mode. The first procedure, `collect(str:string)`, prepends a single string to the `strings` resource. In this procedure, we use the resource by referencing the variable of the same name.

The second procedure, `collect(tree:tree(string))`, traverses a defined `tree` type. We begin by deconstructing the `tree` if it is of the `node` constructor, into a `left` tree, the node's value, `str`, and a `right` tree. We then recursively `collect` the strings from the `right` tree, append the value `str`, and recursively collect the strings from the `left`

```
# declare `strings` resource, a list of strings
resource strings:list(string)

type tree(T) { pub nil | node(left:tree(T), value:T, right:tree(T)) }

# prepend a single string to the head of `strings`
def collect(str:string) use !strings {
    ?strings = [str | strings]
}

# add all strings from the tree to the start of `strings`.
# traverse right-to-left to ensure the strings are accumulated in-order
def collect(tree:tree(string)) use !strings {
    if { tree = node(?left, ?str, ?right) ::
        !collect(right)
        !collect(str)
        !collect(left)
    }
}

# create a scope where we can use `strings`
use strings in {
    # bind `strings` to the empty list
    ?strings = []
    # construct a tree
    ?tree = node(node(nil, "a", nil),        # tree:   "b"
                 "b",                        #        /   \
                 node(nil,                   #      "a"   "c"
                     "c",                    #               \
                     node(nil, "d", nil))))  #               "d"
    # collect strings from the tree
    !collect(tree)
    # after: strings = ["a", "b", "c", "d"]
}
```

Listing 1.8: Example resource and `use` block usage.

tree. In this procedure we do not refer to the resource by name, instead the state of the resource is passed implicitly.

In the top-level code, we create a scope where we can `use` the `strings` resource. Inside this scope the `strings` resource can be used, but it is not bound to a value. We bind the `strings` resource to the empty list `[]`.

We then `collect` all strings from a `tree` structure. This results in the `strings` resource having a value of `["a", "b", "c", "d"]`, which is a left-to-right ordering of the

elements of the `tree`.

1.3.6 Determinism

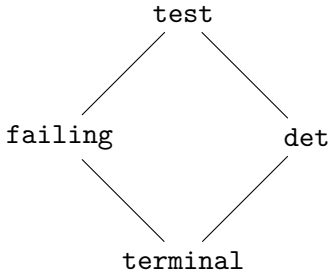
In Wybe, the determinism of a procedure is a property of a procedure that dictates two properties: the number of solutions of a procedure, and if the procedure can fail. Determinisms allow for the programmer to declare where a procedure may fail to produce a result, with this property being used by the compiler to generate code that handles such failures appropriately.

The determinism system of Wybe is similar to other languages, such as Mercury, however, differs in its current state, lacking the `nondet` and `multi` determinisms. In Wybe, there are four determinisms, as outlined in Figure 1.1. The properties of each determinism are checked by the compiler to ensure correctness.

The different determinisms of a procedure call are valid in different determinism contexts. For a context to be considered to have a determinism, it may be that the context is inside a procedure with said determinism, or in the case of a conditional statement's condition, the context has the `test` context.

		Maximum number of solutions	
		0	1
Can fail?	No	`terminal`	`det`
	Yes	`failing`	`test`

(A) Properties of determinisms.



(B) The determinism lattice.

FIGURE 1.1: The determinism system of Wybe.

By default, a procedure has the `det` determinism. Such a procedure behaves much like procedures in regular languages. That is, the procedure cannot fail and succeeds exactly once. `det` procedure can be called in any context.

The `test` (or `partial`) determinism denotes a procedure that `may` fail to produce a result. `test` procedures are valid in both `test` and `failing` contexts. An example `test` call can be found in Listing 1.9.

Optionally, a `test` procedure call can be called with an out-flowing Boolean-typed variable, reifying the `test` procedure call into a `det` procedure call. This variable is true if the procedure succeeded, otherwise, it is false. Conversely, if the final parameter of a `det` procedure call is an output with a Boolean type, and the corresponding argument is omitted, the `det` procedure call can be de-reified into a `test` procedure call.

```
1 def {test} even(n:int) { n % 2 = 0 }
2
3 # if condition is a test
4 if { even(3) :: !println("that's odd!") }
5
6 # reified test
7 even(4, ?succeeded)
8 if { succeeded :: !println("that's even!") }
```

LISTING 1.9: Example `test` procedure, usage and reification.

The `failing` determinism denotes a procedure that will always fail, providing zero solutions. A built-in `failing` procedure is the `fail` procedure, which takes zero arguments and fails. `failing` procedures are valid in both `failing` and `test` contexts.

The final determinism in Wybe is the `terminal` determinism. A `terminal` procedure is a procedure that will never succeed nor fail, providing zero solutions. This means that, when a `terminal` procedure is called, all subsequent statements will not execute. A `terminal` procedure can be called in any context. Example `terminal` procedures include infinite loops and procedures that can exit the program such as the `error` procedure.

1.3.7 Purity

Purity is the property of a procedure or function to perform identically when provided identical inputs and to have no observable side effects. A pure procedure is said to be an analogue of a mathematical function, and knowledge of a procedure's purity is of interest internally to the compiler. As a pure procedure behaves identically on identical inputs, multiple invocations of the same pure procedure are subject to common sub-expression elimination. Common sub-expression elimination allows a previously computed output of some previous invocation of the same procedure with identical inputs to be reused, eliminating the successive call to the pure procedure.

All Wybe procedures are `pure` by default and can call other `pure` (and `semipure` procedures). `pure` procedures are subject to call re-ordering, call omission, and common sub-expression elimination. All calls to non-pure (`semipure` or `impure`) procedures are marked by a preceding `!`, as shown in Listing 1.10. This is to remind the reader that the call is not pure.

```
1 def {semipure} update_string(s:c_string, i:int, c:char) {
2     foreign lpvm {impure} mutate(s, ?s, i, 1, 1, 0, c)
3 }
4
5 # create a C-style string
6 ?str = c_string("abc",,"xyz")
7 !update_string(str, 1, 'B')
```

Listing 1.10: Example non-pure procedure and call.

Each procedure can have one of three purity modifiers: `pure`, `semipure`, and `impure`. An `impure` procedure is free to call procedures of all purity levels. A `semipure` procedure has the property that it is allowed to call all levels of purity, however, is subject to call-reordering, a property that `impure` procedures do not have. Like regular pure procedures, procedures with an explicit `pure` modifier can be subject to call re-ordering, call omission, and common sub-expression elimination, however, are allowed to call `impure` procedures within its body.

These optimisations are available for pure procedures due to the property that a pure procedure acts as a mathematical function. Pure procedures have no side effects, always producing the same outputs with identical inputs. Call-reordering allows for re-ordered calls to be optimised differently, exploiting redundancies that allow for other optimisations to produce more optimised code. With Wybe's copy-on-write semantics, re-ordering may allow for these semantics to be violated if re-ordering produces equivalent code with un-aliased memory. An example of when this optimisation can and cannot be performed with re-ordered pure code can be found in Listing 1.11.

```
1 reverse(ls, ?rev)
2 sum(ls, ?ls_sum)
3 sum(rev, ?rev_sum)
```

(A) `rev` cannot destructively re-use the allocated memory of `ls`, as `ls` is used after.

```
sum(ls, ?ls_sum)
reverse(ls, ?rev)
sum(rev, ?rev_sum)
```

(B) `rev` can destructively re-use the allocated memory of `ls`.

LISTING 1.11: Example calls that, when re-ordered, can produce more optimal code.

1.4 Current Compiler Architecture Overview

The Wybe compiler can broadly be decomposed into three major stages. These stages correspond to an intermediate representation of a given program: the AST, LPVM, and LLVM.

The first stage represents a Wybe program in an AST form, being an abstract representation of the syntax of the Wybe language. This component performs manipulation of the source code, performing tokenisation and parsing, producing a denormalised AST. The denormalised AST is then normalised, introducing implicit procedures for declared types (constructors, deconstructors, getters, setters, *etc.*) and transforming function definitions into equivalent procedure definitions. Normalisation also includes a flattening pass of the AST, reducing complex AST constructs such as `for` loops into simpler, semantically equivalent, forms.

This stage proceeds with type and mode checking. Type and mode checking in tandem are used to disambiguate overloading, resolving which instance of a procedure the call corresponds to. This stage attempts to infer the type of any terms that are not explicitly types and checks that all types are correct in a given program. Mode checking is responsible for the final overloading resolution after all terms have been typed, and for ensuring that variables and resources are appropriately bound before use. Resources are then transformed into positional parameters and arguments in the following pass.

The next pass of the compiler performs more aggressive flattening called *unbranching*. Here the compiler transforms the AST to a restricted form that only contains procedure calls and tailing conditional branches, transforming loops into recursive procedures. All calls after unbranching are also transformed into equivalent deterministic calls, by reifying calls that are not deterministic with the following branch. After unbranching,

the AST resembles LPVM, and as such we perform a pass that transforms the AST into LPVM.

Following the transformation into LPVM, we enter the second major stage. This stage is concerned with the analysis and transformations of the LPVM intermediate representation of a Wybe program. The primary optimisation pass performed on LPVM performs inlining, building upon the general optimisation framework built into the compiler, the *body builder* which performs amongst other optimisations value propagation, and neededness analysis.

This pass also performs alias analysis, inferring which variables alias the same data. This analysis is used to transform, when possible, calls that have certain un-aliased arguments. Due to Wybe's copy-on-write semantics, a program can perform redundant copies on certain writes where the old copy is unused. These transformations allow for code to be transformed to violate the copy-on-write semantics due to this redundancy, however, retaining the same operational semantics. This analysis is also used to perform compile-time garbage collection if memory is un-aliased and unused, then can be reused by some later memory allocation [3].

Finally, we leverage the LLVM compiler [4] to perform the final code generation. We also make use of the optimisation passes implemented in the LLVM compiler, further producing more optimal code. This ultimately produces the object files, used for more efficient incremental compilation [5], and binary executable files.

Chapter 2

Literature Review

2.1 Higher-Order Programming

Higher-order programming refers to the paradigm of programming that makes use of higher-order terms. In contrast to first-order terms, where each value is a concrete value, higher-order programming promotes functions to first-class members of the language. A function term can be called with arguments, as in the case of a regular function call. However, a higher-order call can be made to a variable function.

In Listing 2.1, we see an example of some Haskell code that makes use of higher-order programming. The term `f` in the definition of `map` is a function value, which is applied to all values inside a list, recursively. The terms `ls0` and `ls1` are then defined with the use of the `map` function, applying a defined procedure, `add10`, and a lambda function to all values within the list `[1, 2, 3]`.

```
1 map :: (a -> b) -> [a] -> [b]
2 map f []     = []
3 map f (x:xs) = f x : map f xs
4
5 add10 :: Int -> Int
6 add10 x = x + 10
7
8 ls0 = map add10 [1, 2, 3]
9 ls1 = let y = 10 in map (\x -> x + y) [1, 2, 3]
```

Listing 2.1: Example higher-order program written in Haskell.

Higher-order programming is a hallmark of the functional programming paradigm, yet is increasingly prominent in other paradigms, such as imperative programming. Early high-level functional programming languages, such as LISP [6], paved the foundations for later implementation in other languages.

Programming languages can broadly be divided into two categories, depending on the features of the language which support functions as a value. Languages that support functions as second-class citizens allow for higher-order programming through the use of function pointers or related features. In contrast to this, a language may feature functions as first-class citizens of the language, which more naturally represent functions in the language without the use of pointers.

Languages with first-class functions, in conjunction with nested or anonymous function declarations, naturally allow for the introduction of *free variables*, *i.e.*, variables that are defined out of the scope of the definition. An example of a free variable is the variable `y` in Listing 2.1. The lambda function makes use of `y`, which is defined in the parent scope of the definition of the lambda function.

2.1.1 Implementation

In early compilers for functional programmers, lambda lifting was a transformation used, transforming a nested set of function definitions into a set of global function definitions [7]. This transformation is performed in two parts: first, by eliminating free variables by adding parameters to any nested definition, and next by *lifting* nested definitions into the global scope. This transformation also requires all call sites to be adjusted to ensure the previously free variables are passed as arguments. In modern compilers, this transformation is used in conjunction with other techniques [8]. The reverse transformation to lambda lifting is lambda dropping, reducing the scope of a function call, allowing for simpler analysis due to the decreased scope [9].

An alternative to lambda lifting is called closure conversion [10, 11]. Closure conversion is the process of transforming nested function definitions that have free variables into closures. A closure is a representation of such a function definition, being a pair consisting of a function (via a function pointer or some other reference) and a reference to

the *environment* where the function is defined which contains the free variables. Similar to lambda lifting where the compiler hoists function definitions to the global scope, the compiler hoists all nested function definitions to the global scope, but instead, the compiler adds a single additional parameter containing a reference to the environment.

Complications associated with the implementation of first-class functions arise in two distinct forms of the *funarg* (function argument) problem [12, 13]. The upwards funarg problem arises from returning a function from some function call; the downwards funarg problem arises from passing a function as a parameter to a function call. For closures, references to a static environment lead to these exact problems.

The upwards funarg problem arises from the typical usage of stack frames or activation records used to store the local state of all variables in a function call. When a function is returned from, the stack space should become unused, and hence can be *popped*. However, if a function is returned, there may still be references to free variables of the function that were statically allocated on the stack frame where this function was defined.

The downwards problem arises as a dual of the upwards funarg problem. In tail-call elimination, stack frames can be reused if a (recursive) call is made in the tail position. As this is the final call that uses the stack frame, the currently allocated stack frame can be reused as no references to the stack frame will be required again. However, if a function term is defined with references to variables defined on the stack frame, and the tail call makes use of this function term, these variables would be referenced again, invalidly.

One solution to these problems is to make such possibilities impossible. This is done, for example, in the Pascal language, where functions are not allowed to be used as return values, and as such, an implementation of Pascal must only consider the downwards funarg problem. Another solution is to allocate stack frames on the heap, relying on garbage collection or some other form of memory management to clean up the allocated memory when there are no references to local variables. Historically this has been seen as less than favourable due to the overheads commonly associated with heap-based memory allocation, however, there has been doubt cast upon this assumption [14, 15].

Instead of using a static reference to the environment for a closure, the upwards and downwards funarg problems can be solved by allocating stack frames on the heap. Heap allocation may then require the use of some system of garbage collection to reclaim the allocated memory once all references to the stack frame are no longer required. Traditionally this has been seen as having the potential for slowdown, however, has been shown to not be an issue [14, 15]. Some implementations of closures employ a hybrid approach for the allocation of stack frames. Through static analysis, the compiler may deduce that a closure would create no upwards or downwards funarg problems. In this case, the compiler may use statically allocated stack frames. Otherwise, the compiler may still use heap-based allocation.

A further alternative is to copy the values of free variables at the time of the closure's creation into a separate structure. In languages that forbid mutation, such as ML [16], as the value of a variable remains unchanged there is no semantic issue with copying these variables directly. In a language that allows mutation such as Java, however, mutation may cause two copies of variables to diverge as there is no shared state with the closure environment. For instance, in Java, the solution to this problem is to only allow variables to be free if they are constant (declared as `final`).

The representation of the environment of a closure is manifested in two forms: nested and flat. Nested environments are suitable in instances where the environment is a nested set of stack frames. With such a structure, it may be required to traverse this nested structure to find the value associated with a variable. The alternative to a nested environment is a flat environment, which represents all free variables in a vector, requiring a single indirection to attain the value associated with a free variable. Nested environments allow for simpler reuse of environments, possibly saving space over a flat representation. As such, we must consider the trade-offs associated with each form.

Due to the (possible) decrease in execution time, effort has been made to save space with a flat environment representation [17], with a formal proof of the safety of these techniques [18], ensuring good asymptotic behaviour. The ability to share environments can decrease the memory footprint of closure. For instance, if some collection of closures reference the same free variables, the compiler may share the same environment for these closures. Even in the case where some closures may share some of the same free variables, the compiler may choose to create a larger flat environment that contains all

free variables, and reference only those necessary for each closure. Other techniques have been developed to further save space with closures when the number of free variables is small or the function reference is known [19].

2.2 Type Systems

A type system is a set of rules that define how objects in a language are types and how these objects can legally be defined under the type system to create well-typed, ultimately aiding the programmer in writing well-formed programs. These well-formed programs are said to be correctly typed. [20]

Broadly, the type systems employed in various programming languages can be divided into various categories based on certain properties these systems achieve. The type systems of some languages perform these checks at compile time, whereas others perform these checks at runtime. Type systems that perform these checks at compile time are said to be statically type checked; type systems that perform these checks at runtime are said to be dynamically type checked [20]. Static analysis provides stronger guarantees of the correctness of programs before execution, though this can come with a more complex compilation process. Static analysis can further be used to ensure the safety and correctness of compiler transformations and optimisations. The Wybe language utilises a static type system.

Primitive type systems, such as the simply typed lambda calculus ($\lambda^{\rightarrow}$) [21], provide a basis upon which more rich type systems can be derived and extended. The types described by $\lambda^{\rightarrow}$ are described in Figure 2.1, and describe a type system for the lambda calculus, extending the grammar of the lambda calculus to contain type annotations.

$$\begin{array}{lll} \langle \mathrm{t} \rangle & ::= & x \mid \lambda x : \langle \mathrm{T} \rangle \ . \ \langle \mathrm{t} \rangle \mid \langle \mathrm{t} \rangle \ \langle \mathrm{t} \rangle \\ \langle \mathrm{T} \rangle & ::= & C \mid \langle \mathrm{T} \rangle \rightarrow \langle \mathrm{T} \rangle \end{array}$$

FIGURE 2.1: The grammar of $\lambda^{\rightarrow}$.

The types in $\lambda^{\rightarrow}$ are *Curried*. Types of the form $T \rightarrow T$ are function types in the type system. In contrast to functions of differing arity, all functions in $\lambda^{\rightarrow}$ have arity one. Currying provides a simpler function type, leading to simpler theoretical models.

Higher arity function types can be *Curried*, transforming a type such as $(X \times Y) \to Z$ into $X \to (Y \to Z)$ through a process called Currying. Curried functions can represent all higher arity functions through this Currying process, or repeated application of it.

To check that a program in the lambda calculus is correctly typed, or type checks, a series of rules are used to check that each term is correctly typed within the typing context, Γ. It is said that t has type T if $t : T \in \Gamma$, or equivalently, $\Gamma \vdash t : T$. The typing rules of $\lambda^{\to}$ are outlined in Figure 2.2.

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

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

Figure 2.2: The typing rules of $\lambda^{\to}$.

In Figure 2.2, these rules are read as natural deductions. The premises (terms above the horizontal rule) are used to prove the conclusions (terms below the vertical rule). Given the set of premises can be proven, the conclusions can be inferred. These rules are applied to a term and recursively to sub-terms to prove (or disprove) a valid typing.

For instance, consider the type checking of the term $f\ (g\ y)\ x$, with the typing context $\Gamma = \{f : \mathit{Int} \to \mathit{Float} \to \mathit{Int}, g : \mathit{Int} \to \mathit{Int}, x : \mathit{Int}, y : \mathit{Float}\}$. It can be proven that the term has type Int as shown in Figure 2.3.

$$\frac{\dfrac{g : \mathit{Int} \to \mathit{Int} \in \Gamma}{\Gamma \vdash g : \mathit{Int} \to \mathit{Int}}\ \text{T-Var} \qquad \dfrac{x : \mathit{Int} \in \Gamma}{\Gamma \vdash x : \mathit{Int}}\ \text{T-Var}}{\Gamma \vdash g\ x : \mathit{Int}}\ \text{T-App}$$

$$\frac{\dfrac{\dfrac{f : \mathit{Int} \to \mathit{Float} \to \mathit{Int} \in \Gamma}{\Gamma \vdash f : \mathit{Int} \to \mathit{Float} \to \mathit{Int}} \qquad \Gamma \vdash g\ x : \mathit{Int}}{\Gamma \vdash f\ (g\ x) : \mathit{Float} \to \mathit{Int}} \qquad \dfrac{y : \mathit{Float} \in \Gamma}{\Gamma \vdash y : \mathit{Float}}}{\Gamma \vdash f\ (g\ x)\ y : \mathit{Int}}$$

Figure 2.3: Type checking the term $f\ (g\ x)\ y$ in $\lambda^{\to}$, proving the typing of Int in the typing context $\Gamma = \{f : \mathit{Int} \to \mathit{Float} \to \mathit{Int}, g : \mathit{Int} \to \mathit{Int}, x : \mathit{Int}, y : \mathit{Float}\}$.

Some type systems can provide type inference. Type inference allows types of terms within a language to be inferred automatically without annotation in the source code. The simply typed lambda calculus has a type inference algorithm that was also introduced by Church [21].

Extending the type system of $\lambda^{\rightarrow}$ can be performed in 3 primary ways. These extensions form the lambda cube [22], which extends the type system of $\lambda^{\rightarrow}$ in various ways. The axes of the cube correspond to generalisations of $\lambda^{\rightarrow}$ with respect to dependent types (λP), polymorphism (λ^2) and type operators ($\lambda\underline{\omega}$).

The λ^2 type system (also known as System F) [23, 24] introduces polymorphism to the type system. Polymorphism promotes types as parameters to other types, allowing universal quantification of such types. Type inference in System F has been shown to be undecidable [25].

A restriction to System F, known as the Hindley-Milner type system [26–28] restricts where type quantifiers can occur in the type system. Whereas in System F where type qualifications can occur anywhere in a type, type quantifiers exist only at the prenex (top-most position) of types. With this constraint on type quantifiers, the Hindley-Milner type system does have a decidable type inference algorithm, Algorithm W. As such, the Hindley-Milner type system is a popular type system, and was first implemented in the ML family of languages [16].

Various extensions of the Hindley-Milner type system also exist. One such example is the type class system used as an extension of the Hindley-Milner type system in the Haskell language [29]. Type classes provide constraints on quantified types that allow for a system that is similar to an interface system as seen in object-oriented languages. Other extensions to System F include System $F_{<:}$, which features subtyping [30].

There has been numerous attempts to add static type systems to logic programming languages, such as Prolog [31–33]. These type systems, however, have failed to gain traction in the Prolog user space due to this being an extension to the Prolog language, or being incompatible with conventional Prolog programs. The Mecrcury language [34], a logic programming language closely related to Prolog, does have a static type system, that is similar to that of the Hindley-Milner type system, though is founded upon many-sorted logic [35].

2.3 Intermediate Representations

Internally in a compiler, a program is represented in a form known as an intermediate representation. Intermediate representations are designed primarily to represent a program in a representation that allows for analysis of the representation and subsequent transformations. Ideally then an intermediate representation should be designed to facilitate both analysis and transformations, allowing for not only an efficient representation but a flexible and expressive representation [36].

2.3.1 Static Single Assignment and Allied Forms

A common property of intermediate representations is the static single assignment form (SSA form). SSA form was initially developed by Rosen et al. [37] for redundancy elimination in programs, removing redundant instructions that have already been computed or are provably equivalent to some other computation. Prior techniques were limited in the scope of their analyses, and hence transformations, however, with SSA form, these analyses were able to extend the scope of analysis from being confined to a single basic block to a global analysis.

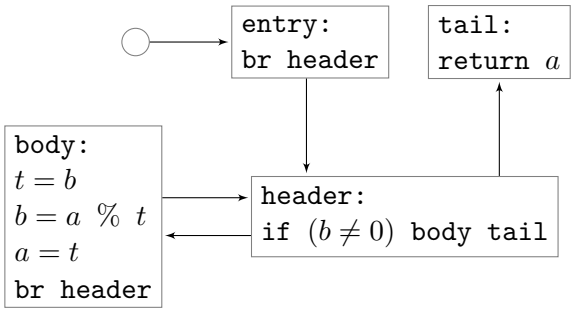
SSA form was later popularised after an efficient algorithm was devised by Cytron et al. [38], which can transform other intermediate representations into SSA form. This algorithm utilises *dominance frontiers* [39, 40], which are used to compute which definitions dominate other definitions, allowing for ϕ-nodes to be placed only where dominance frontiers exist.

While SSA form is not limited to certain representations, the SSA form property is typically associated with control flow graph-based representations, which represent programs as basic blocks (a list of instructions with no intervening branches) forming the nodes of the control flow graph, and branches or jumps forming the edges connecting the nodes appearing at the end of each basic block. These intermediate representations are naturally suited for imperative programming languages.

SSA form requires each variable to have a single assignment. In transforming some intermediate representation into SSA form, one must distinguish different assignments

```
1  int gcd(int a, int b) {
2      while (b != 0) {
3          int t = b;
4          b = a % t;
5          a = t;
6      }
7      return a;
8  }
```

(A) The GCD program written in C.



(B) The GCD program translated into a control flow graph.

FIGURE 2.4: A GCD program written in C, with a corresponding control flow graph representation of the program. [41]

to variables of the same name. This is performed by transforming each assignment to the same-named variable into an assignment to a numbered version of the variable.

However, depending on which block was the predecessor of some block, a different version of each variable may have been assigned to. To amend this, ϕ-nodes are introduced at the beginning of each block. For each variable that may have been assigned a different version in some incoming block, the ϕ-node associated with this variable assigns a fresh version of this variable, which is then used subsequently in the rest of the block, disambiguating which version of each variable currently exists. These ϕ-nodes are either assembled into a no-op, or a move instruction if register allocation fails to allocate the versions to the same register, in the final passes of code generation. In essence, ϕ-nodes are not real instructions, however, are necessary to maintain the restrictions of SSA form in a control flow graph.

SSA form uses ϕ-nodes to encode the information of *def-use chains*. A def-use chain is a mapping of definitions (variable assignments) to their usages [42]. As a variable may be assigned numerous times, the definition of a variable may not necessarily reach some use of the variable. Encoding this information with ϕ-nodes allows for a compact representation of these chains, as each (version of a) variable is assigned exactly once.

Noting that each variable is assigned exactly once in Figure 2.4, the control flow graph from Figure 2.5 has been transformed into SSA form. In the `header` basic block, we have two ϕ-nodes, one for the a variable, and one for b. The ϕ-node for variable a semantically means that, if execution entered via the `entry` block then a_0 is assigned

the value of a, else if execution entered via the `body` block then a_0 is assigned the value of a_1. This is likewise with the value of b_0, assigned to the value of b or b_1, respectively.

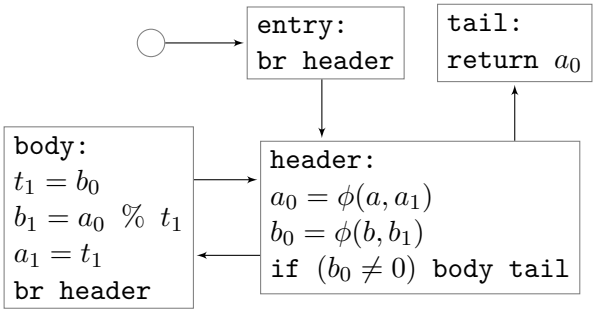


FIGURE 2.5: The GCD control flow graph in SSA form. [41]

The algorithm that introduces ϕ-nodes into a control flow graph, however, can also introduce some extraneous ϕ-nodes into the SSA form control flow graph. To this end, considerable effort has been made to reduce the number of ϕ-nodes, with a focus on producing a control flow graph with the minimal number of ϕ-nodes possible [43, 44]. Reducing the number of ϕ-nodes in the control flow graph allows for more efficient analyses and transformations, as ϕ-nodes bloat the intermediate representation and inhibit certain forms of analysis.

There exists a considerable downside to the use of ϕ-nodes in a control flow graph, though their use is imperative to the SSA form property. A ϕ-node introduces a forward bias in the graph, impeding analyses that work backwards against the flow of execution through the graph.

To amend the forward bias introduced via the inclusion of ϕ-nodes, a dual to the ϕ-node, σ-nodes, are introduced [45], forming static single information form (SSI form). In contrast to ϕ-nodes, which are introduced at the start of basic blocks, σ-nodes are introduced at the end of a basic block. Semantically, a σ-node is used to represent which blocks this block's variables are used in. The introduction of σ-nodes allows for analysis in a backward direction, which in conjunction with ϕ-nodes allows for analysis, and hence transformations, in both forward and backward directions.

ϕ-nodes, however, still have limitations with some forms of analyses, even with the extensions of SSI form. In non-relational analyses, abstract values can propagate through ϕ-nodes by taking the join of each abstract value for each input variable version. However, in relational analyses, there is no mechanism to propagate information about the relationship between two or more variables, such as knowledge of the equality or ordering

of certain variables. A remedy for this would have to reconsider the notion of a ϕ-node to consider the value of potentially all other variables, to provide the level of analysis required for relational analysis.

Further additions have also been made to SSA form intermediate representations. Gated single assignment form (GSA form) [46] introduces γ-nodes. Similar to ϕ-nodes, γ-nodes are intended to merge different versions of the same variable at some merging point in the control flow graph. However, unlike ϕ-nodes, γ-nodes require an additional predicate argument. This predicate argument defines which branch execution reached the current block via. As an extension to GSA form, thinned-gated single assignment form (TGSA form) introduces both μ-nodes and η-nodes, which represent loop headers and loop exits, respectively. In SSA form, ϕ-nodes can be used in place of both γ-nodes and μ-nodes, while η-nodes have no counterpart in SSA form.

With these additional nodes, GSA and TGSA allow for analyses that incorporate constraints based on the information contained within the predicates of these additional nodes. However, GSA and TGSA do not introduce mechanisms that allow for backwards analyses as was possible in forms such as SSI. These additional nodes also further complicate the intermediate representation.

2.3.2 Continuation Passing Style

In contrast to the control flow graph-based intermediate representations typically present in the compilers of imperative languages, functional languages typically utilise a declarative intermediate representation. Continuation passing style (CPS) was first used to encode control flow by Strachey [47], however, was first used as an intermediate representation in a compiler for the Scheme language [11]. CPS naturally represents constructs from functional languages, such as closures.

In CPS, control flows via continuations, additional parameters that replace the `return` instructions with a function that takes the return value. Continuations then use the returned value as a parameter to the rest of the computation. For instance in Figure 2.6, the parameter `k` is a continuation, and is used in place of a `return` instruction to pass the value of `a` on through the continued computation.

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

FIGURE 2.6: The GCD program in CPS style.

In contrast to control flow graph-based intermediate representations, there is no need for name management modifications, such as introducing ϕ-nodes to merge different versions of the same variable. This is due to CPS passing variables to other functions, which themselves serve as the basic blocks of the intermediate representation.

SSA form is equivalent to a subset of CPS which excludes non-local control flow [48, 49]. Non-local control flow, however, is not typically used when employing CPS, nor introduced through conventional program transformations, and as such, the two can be considered equivalent. Due to this, CPS must also suffer from the forward bias that is present in SSA form intermediate representations.

Higher-order programming constructs have been associated with computational overheads, such as the creation and calling of closures. Certain intermediate representations have been designed and implemented with a focus on higher-order specific optimisations, attempting to eliminate some overheads associated with higher-order programming. One such example is the Thorin intermediate representation [8], which itself is based on CPS intermediate representations. Thorin makes use of a novel transformation called lambda wrangling. Lambda wrangling is a combination of lambda dropping and lambda lifting and has been demonstrated to subsume various program transformations, including tail-recursion elimination and loop unrolling.

The optimisations made available through the lambda mangling transformation have been shown to provide a speed-up in the resultant program's execution time across a suite of benchmarks. This eliminates most of the overhead associated with closures, matching the performance of equivalent C programs within a small margin.

2.3.3 Logic Programming Form

As introduced by Gange et al. [41], logic programming form (LP form) intermediate representations utilise Horn clauses as an intermediate representation for program analysis and transformation. Horn clauses [50] are used to represent programs as a series

of logical clauses (rules) which are used to make logical deductions to perform the represented computation. Horn clauses are used as a model of computation in many logic programming languages, such as Prolog [51] and Mercury [34]. LP form is 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. LPVM is the primary intermediate representation in the Wybe compiler.

LP form intermediate representations represent a procedure as a goal together with a collection of clauses. For a given goal, the collection of clauses has the property that, for any given input, exactly one clause can ever succeed.

Clauses also handle name management similarly to how CPS handles name management, lacking an explicit `return`, and passing variables to other blocks (goals) via parameter passing. However, unlike with CPS, there is no explicit continuation parameter. Instead, LP form intermediate representations allow for variables to be passed in and out (as though returned from) some clause. Variables in LP form also maintain the single assignment property, being assigned once in a given procedure. Thus, LP form languages are considered to be in SSA form. Unlike SSA form control flow graphs, though, LP form intermediate representations do not need explicit ϕ-nodes in the intermediate representation to handle name management. This also simplifies the recognition of def-use chains in an LP form language, as all assignments will reach every usage of each variable.

Paralleling branches seen in other intermediate representations, LP form intermediate representations utilise guards. The clauses of a given procedure, pairwise, are identical up to some Boolean guard. After this they *fork*, diverging from the guard onwards. Guards are constructed such that all clauses are mutually exclusive and complete, ensuring the property that for any given input a single clause will succeed. This tames the non-determinism seen in typical use cases of Horn clauses (as in Prolog and Mercury), allowing for deterministic evaluation. Unconditional branches are paralleled with terminal procedure calls, and loops are naturally represented through recursion.

Noting that there is redundancy in the intermediate representation up to a common guard, implementations of LP form languages do not represent clauses as a set of clauses but as a tree. This tree is manifested as a list of goals that are common in all clauses,

ending in an optional guard. If the guard is present, subsequent goals are present in multiple forks of the tree. This provides a more convenient representation of an LP form intermediate representation, as the analysis of common goals in a clause can be performed together without the requirement to consistently transform numerous clauses.

$$
\begin{aligned}
\mathtt{gcd}(\mathtt{a},\mathtt{b};\mathtt{r}) &\coloneqq \mathtt{gcd}_{\mathtt{header}}(\mathtt{a},\mathtt{b},\mathtt{t};\mathtt{r}) \\
\mathtt{gcd}_{\mathtt{header}}(\mathtt{a},\mathtt{b},\mathtt{t};\mathtt{r}) &\coloneqq \mathtt{gcd}_{\mathtt{guard}}(\mathtt{a},\mathtt{b},\mathtt{t};\mathtt{r}) \\
\mathtt{gcd}_{\mathtt{guard}}(\mathtt{a},\mathtt{b},\mathtt{t};\mathtt{r}) &\coloneqq \mathtt{b} = 0 \ \wedge \ \mathtt{gcd}_{\mathtt{body}}(\mathtt{a},\mathtt{b},\mathtt{t};\mathtt{r}) \\
\mathtt{gcd}_{\mathtt{guard}}(\mathtt{a},\mathtt{b},\mathtt{t};\mathtt{r}) &\coloneqq \mathtt{b} \neq 0 \ \wedge \ \mathtt{gcd}_{\mathtt{tail}}(\mathtt{a},\mathtt{b},\mathtt{t};\mathtt{r}) \\
\mathtt{gcd}_{\mathtt{body}}(\mathtt{a},\mathtt{b},\mathtt{t};\mathtt{r}) &\coloneqq \mathtt{t}' = \mathtt{b} \wedge \mathtt{mod}(\mathtt{a},\mathtt{t}';\mathtt{b}') \\
&\qquad \wedge \ \mathtt{a}' = \mathtt{t}' \ \wedge \ \mathtt{gcd}_{\mathtt{header}}(\mathtt{a},\mathtt{b},\mathtt{t};\mathtt{r}) \\
\mathtt{gcd}_{\mathtt{tail}}(\mathtt{a},\mathtt{b},\mathtt{t};\mathtt{r}) &\coloneqq \mathtt{r} = \mathtt{a}
\end{aligned}
$$

FIGURE 2.7: The GCD program in an LP form intermediate representation, as introduced by Gange. [41]

In Figure 2.7 and Figure 2.8 the GCD program is translated into an LP form language. Each procedure call is manifested as the procedure name followed by a two comma separated lists of arguments, separated by a semicolon. The left arguments are the call inputs and the right arguments are the call outputs. Procedures are listed on the left, with each clause being represented individually. Note also that the two clauses of $\mathtt{gcd}_{\mathtt{guard}}$ in Figure 2.7 and the clauses of $\mathtt{gcd}$ in Figure 2.8 are identical up to the guards, $\mathtt{b} = 0$ and $\mathtt{b} \neq 0$, ensuring that exactly one clause applies for each input.

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

FIGURE 2.8: The GCD program in an LP form intermediate representation after simplification. [41]

LP form addresses the aforementioned issues with existing intermediate representations while remaining relatively simple in comparison. As standard with logic programming, clauses are an unordered logical conjunction of numerous goals. This allows for the re-ordering of goals within a clause, however, unconstrained clause re-ordering may destroy the deterministic properties of the intermediate representation.

This further ensures that backwards and forwards analyses and transformations are natural within the representation, being performed with relative ease. As ϕ-nodes,

and derivatives, are not present in the representation, relational analyses can also be performed without the difficulties apparent with the presence of such nodes.

LP form intermediate languages also better model the machine languages of computers. The ability to represent operations with multiple outputs reflects better the capabilities of CPUs than the restriction of prior intermediate representations. For example, the x86 instruction `IDIV` produces a quotient and a remainder in separate registers, and numerous other instructions modify flags in addition to other registers. Without the ability to model multiple return values, the full expressiveness of the architecture cannot be abstracted in the intermediate representation.

Chapter 3

Syntax and Semantics

This chapter introduces the semantics we wish to achieve with higher-order programming in Wybe. The extension aims to extend the semantics of the Wybe language naturally, closely emulating the original, first-order, semantics in the Wybe language. The syntax used in the extension to the Wybe language is also introduced.

3.1 Higher-Order Terms and Calls

A higher-order call is a call to some procedure that is a higher-order term, *i.e.*, a term with a higher-order type. In some languages, such as Prolog and Mercury, the syntax for higher-order calls is verbose, requiring additional procedures, `call/N`, to perform higher-order calls. For Wybe, we wish to keep the syntax concise and align with that of first-order calls, much like in many other languages, such as C and Python. Examples of the syntax of higher-order calls in Wybe can be found in Listing 3.1.

For first-order calls, the call is made to a (possibly module-qualified) name, with a preceding `!` if the call either uses some resource or is non-pure. We continue this syntax with higher-order calls, requiring a preceding `!` if the term is resourceful or non-pure. A higher-order call cannot have a module-qualification, as variables have no module-qualification, existing only in the scope of a procedure.

Each higher-order term has a higher-order type, which is a list of types and accompanying flows. A call to a higher-order term is correct if the type and flow of each argument is identical to that of the higher-order term's types and flows.

```
1  def call_procedure(f:(int, ?int), x:int) use !io {
2      f(x, ?y)
3      !print(y)
4      !print(f(x)) # functional shorthand
5  }
6
7  def call_test(f:{test}(int), x:int) use !io {
8      if { f(x) ::
9          !print("test succeeded")
10     | else ::
11         !print("test failed")
12     }
13 }
```

Listing 3.1: Example higher-order calls in Wybe.

We encapsulate properties of a higher-order type with a set of modifiers, marking the purity, determinism, and resourcefulness (see Section 3.2) of the term. Higher-order calls to terms with these modifiers act as their first-order counterparts.

3.1.1 Anonymous Procedure Syntax

In most languages that support higher-order programming, the language provides a syntax that defines a procedure or function inline. These are commonly referred to as anonymous functions or lambda functions.

The syntax we describe for Wybe is different from that of typical anonymous procedures or functions. This syntax closely emulates the syntax of a procedure definition, however, occurs without an explicit name as a regular procedure declaration would appear, and occurs inline as an expression. This expression is a sequence of several statements composed within braces ({ ... }).

Unlike typical anonymous functions, as seen in languages such as Haskell or Java, the syntax we define for the Wybe language lacks explicit names for the inputs and outputs of an anonymous procedure. Parameters are replaced with *holes*, which is syntactically an `@` sign optionally followed by an integer greater than zero, *e.g.*, `@` or `@2`. All holes in the same anonymous procedure must be either numbered or unnumbered, and a syntax error will be raised if both numbered and unnumbered holes are used in the same anonymous procedure.

The number of a hole dictates the parameter's position, with duplicates referring to the same parameter. The position of unnumbered holes follows an in-order traversal of the source code. Unnumbered holes are a syntactic shorthand and are applicable only in the case that all numbered holes are both used once and in order. An example of equivalent anonymous procedures with numbered and unnumbered holes is seen in Listing 3.2, lines 1–2.

```
1  ?f = { @1 + 1 = ?@2 }
2  ?f = { @ + 1 = ?@ }
3
4  ?g = { ?@2 - 1 = @1 }
5
6  ?h = {
7      ?t = @1 + @2
8      ?@3 = t * (@4 * t)
9  }
10 ?h = {
11     ?t = @ + @
12     ?@ = t
13 }
14
15 ?i = { @1 + @2 = ?@1 }
16
17 ?j = { @1 = ?@3 }
18
19 ?x = 1
20 ?k = { @1 + x = ?@2 }
```

(A) Anonymous procedures.

```
def f(p1, ?p2) { p1 + 1 = ?p2 }

def g(p1, ?p2) { ?p2 - 1 = p1 }

def h(p1, p2, ?p3) {
    ?t = p1 + p2
    ?p3 = t
}

def i(!p1, p2) { p1 + p2 = ?p1 }

def j(p1, p2, ?p3) { p1 = ?p3 }

def k(x, p1, ?p2) { p1 + x = ?p2 }
```

(B) Equivalent procedures.

LISTING 3.2: Example anonymous procedure syntax with equivalent procedures. Where the anonymous procedure is defined twice, the definitions differ in being (un-)numbered.

Each hole is treated as a variable within the anonymous procedure body, and as such can be used with an accompanying mode. The mode of all usages of holes with the same number within an anonymous procedure indicates the mode of the argument. For holes that are used with exactly one mode (*i.e.*, in, out, or in/out), the mode is exactly that, otherwise, it is in/out. If a number is skipped, the mode of the argument with that number is treated as an input and ignored.

Anonymous procedures also can capture the state of the procedure in which they are defined. If a variable is defined in the procedure before the anonymous procedure, this variable can be captured as a *free variable* within the anonymous procedure. An example

of such can be seen in Listing 3.2, line 20, where inside the anonymous procedure bound to `k` the variable `x` is captured as a free variable. The equivalent procedure provided is not a true equivalent as procedure declarations cannot contain free variables.

Upon each invocation of the anonymous procedure, the value of a free variable is the value of the variable when the anonymous procedure was defined. Likewise, any modifications to a free variable are not propagated outside the anonymous procedure. This captures the value of the variable, not a reference to the variable, as is the case in some imperative languages.

An anonymous procedure may be preceded by a collection of modifiers. These modifiers, syntactically placed between braces ({ ... }), allow for various modifiers to be specified for the anonymous procedure, being similar to the modifiers of regular procedure definitions. These specify the determinism (`test`, `failing`, or `terminal`), purity (`pure`, `semipure`, or `impure`) and ability to use resources (`resource`) of the anonymous procedure. If omitted, the anonymous procedure is deterministic, pure, and cannot use resources. For instance, these modifiers allow for an anonymous procedure to be called as a `test` in a `test` context, or an anonymous procedure to call `impure` procedures if marked as `semipure`.

3.1.2 Partial Application

Common in languages with higher-order programming is the concept of partial application. Partial applications allow for some arguments to be specified, producing a term that can be called with the remaining arguments specified. In Wybe, the mode system allows for arbitrary arguments to be specified as outputs. This contrasts with other languages wherein all arguments are inputs and there is a single return value. With partial applications, the arguments that can be specified in a partial application must be inputs as there is no guarantee an output is ever produced.

Partial applications achieve the same result as the use of an equivalent anonymous procedure (Section 3.1.1). However, we choose to also offer partial application as a more compact syntactic representation when the leading argument(s) are the ones to be specified.

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

?add_one = add(1)
?add_one = { add(1, @, ?@) } # equivalently

?three = add_one(2)
```

LISTING 3.3: Example of partial application in Wybe.

3.2 Higher-Order Resources

With the resource system and higher-order programming in Wybe, one naturally is led to ask how higher-order programming and resources can be used together. As procedures in Wybe are annotated with their resource usage, a natural extension of these semantics is one where each higher-order term is annotated with the resources that the higher-order term uses, being part of the higher-order type.

```
def map(f:(X) use !io, xs:list(X)) {
    for ?x in xs {
        !f(x)
    }
}

!map(println, [1, 2, 3]) # print list of ints
```

LISTING 3.4: Potential semantics for higher-order resources, exemplifying the annotations of which resources the higher-order term can use.

Listing 3.4 shows an example of how these semantics may be translated into syntax. The higher-order term `f` in `map` is annotated with the use of the `io` resource, allowing a procedure that uses the `io` resource to be applied to all elements in some list. However, the increased expressiveness gained with higher-order programming is ultimately lost with these semantics. The annotation of exactly which resources the higher-order term can use is detrimental for two factors. First, if we were to write `map` using another set of resources, a very similar definition of `map` has to be provided. Second, the annotation of flow has similar detrimental effects on the verbosity of this semantics.

In Listing 3.5, we exemplify how these higher-order resource semantics decreases the expressiveness of higher-order code. Each definition of `map` requires the annotation of

```
resource counter:int
def map(f:(X) use !io, xs:list(X)) {
    for ?x in xs {
        !f(x)
    }
}
def map(f:(X) use {!io, counter}, xs:list(X)) {
    for ?x in xs {
        !f(x)
    }
}
def map(f:(X) use {!io, ?counter}, xs:list(X)) {
    for ?x in xs {
        !f(x)
    }
}
def map(f:(X) use {!io, !counter}, xs:list(X)) {
    for ?x in xs {
        !f(x)
    }
}
```

LISTING 3.5: Potential syntax for higher-order resources, exemplifying the verbosity of the semantics. Note that the flow of the `counter` resource is different in the latter definitions of `map`.

exactly which resources the term can use, and the flow of each resource. Further, if a new resource is introduced, each definition may be duplicated for each flow of the new resource.

An alternative to this explicit annotation may be to forgo all resource annotations and allow for a higher-order term to freely use any resource. This comes at the cost of the optimisations available for the compiler. Lacking the annotation of resource usage means that any higher-order term may use any resource, and hence optimisations will need to conservatively assume that some resource is modified. Calls to such higher-order terms could also not be annotated with a preceding `!` to mark the call using some resource.

As a compromise between these two semantics, we propose a semantics that annotates the usage of *some* but not which resource(s), by marking each higher-order type with a `resource` modifier if may use said resources. We believe that this semantics allow for the increased expressiveness of higher-order programming while retaining key properties

of first-order resources. The implementation strategy we devise for this semantics can be found in Section 5.2.

```
1  def map(f:{resource}(X), xs:list(X)) {
2      for ?x in xs {
3          !f(x)
4      }
5  }
6
7  !map(println, [1, 2, 3]) # print a list
8
9  resource counter:int = 0
10
11 def tally(x:int) use !counter { ?counter = counter + x }
12
13 !map(tally, [1, 2, 3]) # tally with the counter resource
```

LISTING 3.6: Increased expressiveness with the proposed semantics of higher-order resources.

With the use of the example in Listing 3.6, the term `f` may use some resource. This allows for the same definition, here of `map`, but is independent of which resource(s) the mapping function, `f`, uses. Here we use the same definition of `map` to print a series of elements from a list and tally the sum of a different list of elements. This is analogous to the `mapM_` function in Haskell, which allows a single monadic action to be applied to all elements of a list, however, in Wybe, multiple resources may be used instead.

In this semantics, to ensure that the usage of some resource is sensible, we require that the resource is appropriately in scope where the higher-order term is defined. In Listing 3.6, the `io` and `counter` resources are not in scope in the `map` procedure, however can be modified by the input higher-order term, as these terms, partial applications of `println` and `tally`, were defined in a scope wherein these resources were available and appropriately bound.

For anonymous procedures, the resources the anonymous procedure can use are implicit within the scope where the term is defined. That is, any resource in scope where the anonymous procedure is defined can be used within the procedure. In Listing 3.7, we define numerous anonymous procedures, with a comment indicating if the definition is legal. Legal definitions require that not only resources are appropriately in scope where the anonymous procedure is defined but also are appropriately bound. This is the case

with the first definition of `f` with a call to `res_in`, as the resource `res` is in scope. Note that the resource is not bound after this definition of `f`.

```
1  resource res:int
2
3  def res_in    use  res { pass }
4  def res_out   use ?res { ?res = 1 }
5  def res_inout use !res { ?res = res + 1 }
6
7  use res in {
8      ?f = {resource}{ !res_out }   # legal:   res is in scope
9      ?f = {resource}{ !res_in }    # illegal: res is in scope, not bound
10     ?res = 1
11     ?f = {resource}{ !res_in }    # legal:   res is now bound
12     ?f = {resource}{ !res_inout } # legal:   res is in scope, and bound
13 }
14
15 ?f = {resource}{ !res_inout }     # illegal: res is not in scope
```

Listing 3.7: Example definitions of resourceful higher-order terms.

In the first *illegal* definition of `f`, the call to `res_in` inside the anonymous procedure's body requires that the `res` resource is bound as it is used as input, however this is not the case. As we cannot be sure that the resource has been appropriately bound before the call, this property must be ensured before the anonymous procedure has been defined.

As a higher-order term may not be called, the resource binding state is unchanged. `I.e.`, if some reason is bound to a value inside an anonymous procedure but not bound after before the anonymous procedure is defined, the resource is not bound after the anonymous is defined. The same rules apply to partial applications. Ensuring that resources are appropriately bound and in scope ensures that some parent call scope has the resource in scope. This ensures that the resource is appropriately bound to some sensible value.

`use` blocks provide a scope where some resource is available. With `use` blocks, this may create a scope where some resource is now shared with the caller's scope. Previously, as the resources in scope only affected the resource the procedure was declared to use, `use` blocks introduced a new scope. With higher-order terms, however, a parameter may be able to modify a resource that was not previously in scope.

```
1  resource res:int
2
3  def call(f:{resource}(), ?x:int) {
4      !f()          # call f, may modify res from parent scope
5      use res in {
6          ?res = 1 # assign res
7          !f()      # f may modify res
8          ?x = res # assign x the (possibly modified) value of res
9      }
10     !f()          # as with first call to f
11 }
12
13 use res in {
14     # calling f increments res
15     ?f = {resource}{ ?res = res + 1 }
16
17     !call(f, ?x) # x = 2 after call, res has been incremented twice
18 }
```

LISTING 3.8: Example use of a resourceful higher-order parameter inside a `use` block.

An example of this behaviour can be seen in Listing 3.8. In `call`, the term `f` can use some resource, which may be the same as the resource in the `use` block. The value of `res` is modified both inside and outside the `use` block in the call to `call`, due to the calls to `f` incrementing the resource's value.

Instead of the resource in the `use` block's scope being *different* to that of the resource possibly used by `f`, the semantics we define dictate that the resource modified is the *same* resource. This allows for more interesting interactions, as higher-order terms can interact with the resources in the current scope, and ensures that there is always a single value for each resource.

We believe that the semantics for resources in a higher-order context we have defined in this section attain a balance between the increased expressiveness gained with higher-order programming, while retaining key properties of resources. The explicitness property of resources ensures that *interface integrity* is ensured, as the interface dictates that some resources may be affected. These semantics do relax the constraint of a resource being in scope from the *current* scope to some *parent* scope.

Chapter 4

The Wybe Type System

In this chapter, we introduce the types that exist in the Wybe language and introduce a formalisation of the Wybe type system. We also discuss some implementation details of the type checking algorithm in the Wybe compiler, highlighting the extensions made for higher-order programming, and the constraints on the type system the algorithm places.

4.1 Wybe Types

The Wybe type system has three monomorphic types, *mono-types*. Mono-types are types without any quantifiers. As the Wybe type system is polymorphic, the type system also supports polymorphic types, *poly-types*, with these types present in the prenex position of a type, allowing for type variables to be universally quantified. The syntax of this type system is presented in Figure 4.1. This syntax differs from that found within the Wybe language, with all type variables implicitly universally quantified.

$$
\begin{array}{rcl}
\langle \text{type} \rangle & ::= & \mathit{var} \\
& | & \mathit{module}\texttt{(}\langle \text{type} \rangle^{*}\texttt{)} \\
& | & \texttt{\{}\mathit{mods}^{*}\texttt{\}(}\langle \text{type-flow} \rangle^{*}\texttt{)} \\
\langle \text{type-flow} \rangle & ::= & \langle \text{flow} \rangle\ \langle \text{type} \rangle \\
\langle \text{flow} \rangle & ::= & \mathit{in}\ |\ \mathit{out}\ |\ \mathit{inout} \\
\langle \text{poly} \rangle & ::= & \langle \text{type} \rangle\ |\ \forall \mathit{var}.\langle \text{poly} \rangle
\end{array}
$$

FIGURE 4.1: Types in the extended Wybe type system.

This type system diverges from the types seen in the Hindley-Milner (HM) type system [26–28]. HM requires at least one *type function* to be defined, $\rightarrow$, used to define a function type in the type system.

In the Wybe type system, this function type is replaced by a *procedure type*. Procedure types define the types of procedures, with procedures defined either at the top level or anonymously. A procedure type has a set of modifiers, $\{mods^*\}$, which represent certain properties of the procedure, being the determinism, purity, and the resourcefulness of the procedure. A procedure type's parameters also represent a *type-flow*, a pair consisting of a type and a flow (*i.e.*, *in*, *out* (`?`), *inout* (`!`)).

Before the introduction of higher-order programming in Wybe, procedure types were limited to the types of constant, first-order procedure calls. With the advent of higher-order programming in Wybe, these types can apply to arbitrary variables in the language.

4.2 Wybe Type System

We define the type system as a series of natural deductions in Figure 4.2, Figure 4.3, and Figure 4.4. Natural deductions are read as a series of inference rules, with the premises defined atop the horizontal rule, and the conclusions below. These inference rules are applied recursively to create a proof tree, proving the type correctness of statements and procedures. An example of such a proof tree is found in Section 4.2.4.

These rules are defined with the use of a typing context, Γ. A typing context, Γ, is a collection of variables and corresponding type assignments. That is, the elements of Γ have the form $x : t$, which has the constraint that the variable x has type t. These typing contexts can be extended to include the assignments of more variables, with the notation of $\Gamma, x : t$.

With some typing context, Γ, we indicate the well-typing of some item, be it a statement or definition, or expression, with a turnstile, $\vdash$. *I.e.*, the term $\Gamma \vdash x$ states that x is well-typed under Γ. For expressions, we also indicate the type of the expression under Γ, *i.e.*, $\Gamma \vdash x : t$.

Typing contexts also have a partial order $\Gamma \sqsubseteq \Gamma'$, which states that Γ is a typing that is no less general than Γ'. That is, for all assignments of types in Γ', Γ has an assignment

that is no less general than the assignment in Γ'. The sense of generality is made concrete in Section 4.2.3. This ordering is important in finding a typing context that correctly types multiple statements.

A program is considered to be type correct if all procedures defined in the program are type correct. A procedure is type correct if the body of the procedure, s, a sequence of statements, is type correct under the given typing context, Γ. Upon type checking s, the types of the parameters are extracted and used as the parameter types in the definition. The types of all used resources are checked, ensuring that the inferred type of each resource, $r_i : t'_i$, aligns with the declared type of the resource, as indicated by some resource declaration that states that r_i has type t'_i.

$$\frac{\Gamma, p_1 : t_1, \ldots, p_n : t_n, r_i : t'_i, r_m : t'_m, \vdash s \qquad \forall i.\ r_i : t'_i \text{ resource}}{\Gamma \vdash \texttt{def}\ proc(f_1 p_1 : t_1, \ldots, f_n p_n : t_n)\ \texttt{use}\ r_1, \ldots, r_m\ \{\ s\ \}} \quad [\textsc{Def}]$$

FIGURE 4.2: Typing rules of procedure definitions in Wybe.

4.2.1 Expressions

The Wybe language has three expressions: variables (VAR), constants (CONST), and procedures (PROC), and anonymous procedures (ANONPROC). The typing rules for such expressions are provided in Figure 4.3.

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t} \quad [\textsc{Var}]$$

$$\frac{c : t \text{ constant}}{\Gamma \vdash c : t} \quad [\textsc{Const}]$$

$$\frac{\Gamma \vdash \texttt{def}\ proc(f_1 p_1 : t_1, \ldots, f_n p_n : t_n)\ \{\ s\ \} \qquad t = S\,\mu(f_1 t_1, \ldots, f_n t_n)}{\Gamma \vdash proc : t} \quad [\textsc{Proc}]$$

$$\frac{\Gamma, \forall i.\ @i : t_i \vdash s \qquad \forall i.\ f_i = flow(s, @i)}{\Gamma \vdash \mu\{s\} : \mu(f_1 t_1, \ldots, f_n t_n)} \quad [\textsc{AnonProc}]$$

FIGURE 4.3: Typing rules of expressions in Wybe.

The rule VAR defines the typing of variables. Given the typing context, Γ, if there is some type assignment for the variable x, *i.e.*, $x : t \in \Gamma$, then the system can infer that x has type t, $x : t$.

The Wybe language features several kinds of constants. These include integers, floating-point numbers, characters, strings, and C-style strings. We summarise the typing of constants with the CONST rule. Integers and characters represent a fixed-width integral value, and as such the type of such a constant can be any type where the type has an integral representation. Strings and C-style strings can have only one type, the standard library types, `string` and `c_string`, respectively.

A procedure definition defines a name that can be used to make a call, which can be inferred using the rule PROC. The type of this procedure, $\mu(f_1t_1, \ldots, f_nt_n)$, is defined by the types and flows of the formal parameters of the procedure, $f_1t_1, \ldots, f_nt_n$, and the modifiers, μ, of the procedure, *i.e.*, the purity, determinism, and usage of some resource. Using a substitution, S, type variables present in the procedure's type are uniformly substituted given the substitutions in S. The result of this substitution, $t = S\,\mu(f_1t_1, \ldots, f_nt_n)$ is then inferred as the typing of the procedure. This allows the procedure to be called with differing types across calls to this procedure, enabling polymorphism in the type system.

Anonymous procedures contain a nested body of statements. In the rule ANONPROC, these statements are used to constrain the types of the holes ($@i$) of the anonymous procedure. The modifiers of the anonymous procedure, μ, and the corresponding types, t_i and flows, $f_i = flow(s, @i)$, of each hole $@i$, are used to define the type of the anonymous procedure, $\mu(f_1t_1, \ldots, f_nt_n)$. Γ is also used here to ensure that the typing of variables inside the body of the anonymous procedure is consistent with that of the context the anonymous procedure is defined in.

4.2.2 Statements

The body of a procedure is a sequence of statements. We consider a sequence of statements to be type correct if each statement in the sequence is type correct, as seen in the SEQ rule of Figure 4.4. The sequence, $s_1; s_2$ is then type correct under some typing context, Γ, with the constraint that Γ is no less general than both the typing contexts, Γ_1 and Γ_2, which correctly type s_1 and s_2, respectively.

The remainder of Figure 4.4 defines the typing of each single statement in the Wybe language. Certain statements are always correctly typed, being the `pass`, `break`, and

$$\frac{\Gamma_1 \vdash s_1 \qquad \Gamma_2 \vdash s_2 \qquad \Gamma \sqsubseteq \Gamma_1 \qquad \Gamma \sqsubseteq \Gamma_2}{\Gamma \vdash s_1; s_2} \text{ [Seq]}$$

$$\frac{}{\Gamma \vdash \texttt{pass}} \text{ [Pass]} \qquad \frac{}{\Gamma \vdash \texttt{break}} \text{ [Break]}$$

$$\frac{}{\Gamma \vdash \texttt{next}} \text{ [Next]} \qquad \frac{\Gamma \vdash x : \texttt{bool}}{\Gamma \vdash x} \text{ [TestBool]}$$

$$\frac{\Gamma_c \vdash c \qquad \Gamma_t \vdash t \qquad \Gamma_e \vdash e \qquad \Gamma \sqsubseteq \Gamma_c \qquad \Gamma \sqsubseteq \Gamma_t \qquad \Gamma \sqsubseteq \Gamma_e}{\Gamma \vdash \texttt{if } \{ \ c :: t \mid \texttt{else} :: e \ \}} \text{ [If]}$$

$$\frac{\Gamma \vdash s}{\Gamma \vdash \texttt{do } \{ \ s \ \}} \text{ [Do]} \qquad \frac{\Gamma, r : t \vdash s \qquad r : t \text{ resource}}{\Gamma, r : t \vdash \texttt{use } r \texttt{ in } \{ \ s \ \}} \text{ [Use]}$$

$$\frac{\Gamma \vdash p : t \qquad \exists s = \mu(f_1 t_1, \ldots f_n t_n),\ s \sim t \qquad \forall i.\ (f_i, f_i') \neq (in, out)}{\Gamma, \forall i.\ x_i : t_i \vdash p(f_1' x_1, \ldots, f_n' x_n)} \text{ [Call]}$$

Figure 4.4: Typing rules of statements in Wybe.

`next` statements. These are represented with the Pass, Break, and Next rules, and are correctly typed under any typing context, Γ.

Where a variable, x, has a Boolean-type, `bool`, the variable can be used as a test statement, x. Such a statement is type-correct under some typing context, Γ, if $\Gamma \vdash x :$ `bool`, as defined in the TestBool rule.

Other statements contain nested statements. These statements include conditional `if` statements (If), looping `do` statements (Do), and resource usage `use` blocks (Use). For these statements to be type correct, all nested statements must also be type correct.

Unlike in other languages, the condition in a conditional statement is not required to have a Boolean type, and instead is required to be a correctly typed statement. As such, the type system must check recursively that the condition, c, then branch, t, and else branch, e, are all type correct. The system can then type check the conditional statement with a typing, Γ, that is at least as general as all of Γ_c, Γ_t, and Γ_e. This ensures that all variables across the three nested statements have a uniform typing for all variables.

For `do` statements, if the body, s, of the statement must be type correct under some typing context, Γ, then the `do` statement is type-correct also. A `use` block has the added constraint that in the typing context, the resource, r, has the same type as the

declared resource, t. This ensures that if the resource is referenced by name that the corresponding variable is typed identically to the resource.

All calls must also be type correct as defined by the CALL rule. A call is made to some term, p, of type t, and a sequence of arguments, x_i. For the call to be correct, there exists some type, $s = \mu(f_1 t_1, \ldots f_n t_n)$ that is similar to t, *i.e.*, $s\ t$, where the number of arguments is n, the arity of s. Similarity (defined in Figure 4.5) enables the typing of reified test calls and de-reified tests. Similarity further allows for partially applied calls, with the un-applied parameter's types composing the type of the final output argument, binding this argument to the partially applied term.

$$t \sim s \coloneqq \begin{cases} \forall i \leq n.\ f_i t_i = g_i s_i \ \wedge\ o = ?\texttt{bool} & \text{if } t = \mu(f_1 t_1, \ldots, f_n t_n), \texttt{det} \in \mu, \\ & \quad s = \nu(g_1 s_1, \ldots, g_n s_n, o), \texttt{test} \in \nu \\ \forall i \leq n.\ f_i t_i = g_i s_i \ \wedge\ o = ?\texttt{bool} & \text{if } t = \mu(f_1 t_1, \ldots, f_n t_n, o), \texttt{test} \in \mu, \\ & \quad s = \nu(g_1 s_1, \ldots, g_n s_n), \texttt{det} \in \nu \\ \forall i < n.\ f_i t_i = g_i s_i \ \wedge\ f_n = \texttt{out} & \text{if } t = \mu(f_1 t_1, \ldots, f_n t_n), \\ \quad \wedge\ t_n = \mu(g_n s_n, \ldots, g_m s_m) & \quad s = \mu(g_1 s_1, \ldots, g_n s_n, \ldots, g_m s_m) \\ t = s & \text{otherwise} \end{cases}$$

FIGURE 4.5: Type similarity in the Wybe type system.

The final condition for a call to be type correct is the flow of each argument. The flows of a call are correct if the flow of each parameter, f_i, and the flow of the corresponding argument, f_i', are not *in* and *out*, respectively. From this call, the system is constrained to require that x_i must have the type t_i for the call to be correct.

This condition on the flows of arguments and parameters captures two modes of the call, the direct mode and any *implied modes* [34]. The direct mode occurs when all parameter and argument flows are equal, pairwise, and the execution of the call behaves as defined by the procedure. Otherwise, the call is made with an implied mode, which allows for input arguments in place of output arguments. In an implied mode, this call is now a test, and the execution of this call will succeed if the input arguments are equal to the corresponding outputs of the direct mode call.

4.2.3 Typing Contexts and Orders

Throughout the rules defined prior, the rules make use of an ordering of typing contexts, $\Gamma \sqsubseteq \Gamma'$. We say here that Γ' is a more general typing than Γ. We make the sense of this order clear in this section.

This ordering is used to enable two mechanisms: polymorphism and combining typing contexts between two statements. Polymorphism is enabled through the use of a substitution of type variables from the more general typing.

$$\Gamma \sqsubseteq \Gamma' \coloneqq \exists S \, \forall x : t \in \Gamma'.\ x : S\,t \in \Gamma$$

FIGURE 4.6: Definition of ordering of typing contexts in the Wybe type system

The ordering defined in Figure 4.6 makes use of a substitution, S, which is uniformly applied to all terms that are assigned a typing in Γ. If all type assignments are equal under some such S, then the typing contexts have the relation $\Gamma \sqsubseteq \Gamma'$. This makes use of the unification algorithm defined by Robinson [52], which is also used in the HM type system.

We make use of this ordering to combine various typing contexts, such as in the SEQ rule. In the case of the SEQ rule, the typing contexts, Γ_1 and Γ_2 are combined into one typing, Γ, such that $\Gamma \sqsubseteq \Gamma_1$ and $\Gamma \sqsubseteq \Gamma_2$. This Γ represents a typing context that contains type assignments for all variables in both Γ_1 and Γ_2, however, their type assignments must be unifiable. This unification can be found through the composition of both substitutions.

For example, suppose two statements are type correct under the typing contexts, $\Gamma_1 = \{x : \forall a.\ \texttt{list}(a), y : \forall a.\ a\}$ and $\Gamma_2 = \{x : \texttt{list}(\texttt{int}), z : \texttt{int}\}$. These can be combined to form the typing context, $\Gamma = \{x : \texttt{list}(\texttt{int}), y : \texttt{int}, z : \texttt{int}\}$, under the substitution $S = \{a \mapsto \texttt{int}\}$.

4.2.4 Worked Example

This section provides a short, worked example of the type system. The procedure we will type check is the `gcd` procedure defined in Listing 4.1. This program uses an assignment procedure, `asg`, as the use of the regular assignment statement (of the form `x = ?y`) uses

syntactic sugar, which would complicate the example. The proof is broken into five parts in Figure 4.7.

```
1  def gcd(x, y, ?z) {
2      do {
3          if { nz(y) ::
4              pass
5          | else ::
6              break
7          }
8          asg(y, ?t)
9          mod(x, t, ?y)
10         asg(t, ?x)
11     }
12     asg(x, ?z)
13 }
```

LISTING 4.1: Example GCD program in Wybe.

In this proof, we assume that there are procedures, `mod`, `asg` (assign), and `nz` (non-zero), defined that are type correct, and have types $\{\texttt{det}\}(\texttt{int}, \texttt{int}, ?\texttt{int})$, $\forall \texttt{X}.\ \{\texttt{det}\}(\texttt{X}, ?\texttt{X})$, and $\{\texttt{test}\}(\texttt{int})$, respectively. That is to say that $\Gamma \vdash \texttt{mod} : \{\texttt{det}\}(\texttt{int}, \texttt{int}, ?\texttt{int})$, *etc.*

Throughout the body of the procedure, we will use an instantiation of `asg` to assign `int` values. As the definition of `asg` is generic, having the type $\forall \texttt{X}.\ \{\texttt{det}\}(\texttt{X}, ?\texttt{X})$, and under the substitution of $\{\texttt{X} \mapsto \texttt{int}\}$, we can infer that $\{\texttt{det}\}(\texttt{int}, ?\texttt{int})$, is a valid instantiation of `asg`.

Next, in Figure 4.7A, we know that `nz` has the type $\{\texttt{test}\}(\texttt{int})$, and hence from the call `nz(y)`, we can infer that `y` must have type `int`, resulting in the typing context $\Gamma_{\texttt{nz}}$. As `pass` and `break` are well-typed in all contexts, $\texttt{if}\{\texttt{nz}(\texttt{y}) :: \texttt{pass} \mid \texttt{else} :: \texttt{break}\}$ is also well typed under $\Gamma_{\texttt{nz}}$. Using the instantiation of `asg`, we can infer that $\Gamma, \texttt{t} : \texttt{int}, \texttt{y} : \texttt{int}$ can well type `asg(y, ?t)`. The sequence of these two statements, s_1 is also well typed under the typing $\Gamma_{\texttt{nz}}, \texttt{t} : \texttt{int}$.

Similarly, in Figure 4.7B, we can well type `mod(x,t,?y)` under $\Gamma_{\texttt{mod}} = \Gamma, \texttt{x} : \texttt{int}, \texttt{t} : \texttt{int}, \texttt{y} : \texttt{int}$, and we can use the same instantiation of `asg` to infer that $\Gamma, \texttt{t} : \texttt{int}, \texttt{x} : \texttt{int}$ can well type `asg(t, ?x)`. The sequence of these two calls, s_2, also is well typed under $\Gamma_{\texttt{mod}}$.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \texttt{nz} : \{\texttt{test}\}(\texttt{int})}{\Gamma_{\texttt{nz}} = \Gamma, \texttt{y} : \texttt{int} \vdash \texttt{nz(y)}} \quad \Gamma \vdash \texttt{pass} \quad \Gamma \vdash \texttt{break}}{\Gamma_{\texttt{nz}} \vdash \texttt{if}\{\texttt{nz(y)} :: \texttt{pass} \mid \texttt{else} :: \texttt{break}\}} \quad \dfrac{\Gamma \vdash \texttt{asg} : \{\texttt{det}\}(\texttt{int}, ?\texttt{int})}{\Gamma, \texttt{t} : \texttt{int}, \texttt{y} : \texttt{int} \vdash \texttt{asg(y}, ?\texttt{t)}}}{\Gamma_{\texttt{nz}}, \texttt{t} : \texttt{int} \vdash \texttt{if}\{\texttt{nz(y)} :: \texttt{pass} \mid \texttt{else} :: \texttt{break}\}; \texttt{asg(y}, ?\texttt{t)}}$$

(A) Type checking $s_1 \coloneqq \texttt{if}\{\texttt{nz(y)} :: \texttt{pass} \mid \texttt{else} :: \texttt{break}\}; \texttt{asg(y}, ?\texttt{t)}$ in the `gcd` procedure.

$$\dfrac{\dfrac{\Gamma \vdash \texttt{mod} : \{\texttt{det}\}(\texttt{int}, \texttt{int}, ?\texttt{int})}{\Gamma_{\texttt{mod}} = \Gamma, \texttt{x} : \texttt{int}, \texttt{t} : \texttt{int}, \texttt{y} : \texttt{int} \vdash \texttt{mod(x, t}, ?\texttt{y)}} \quad \dfrac{\Gamma \vdash \texttt{asg} : \{\texttt{det}\}(\texttt{int}, ?\texttt{int})}{\Gamma, \texttt{x} : \texttt{int}, \texttt{t} : \texttt{int} \vdash \texttt{asg(t}, ?\texttt{x)}}}{\Gamma_{\texttt{mod}} \vdash \texttt{mod(x, t}, ?\texttt{y)}; \texttt{asg(t}, ?\texttt{x)}}$$

(B) Type checking $s_2 \coloneqq \texttt{mod(x, t}, ?\texttt{y)}; \texttt{asg(t}, ?\texttt{x)}$ in the `gcd` procedure.

$$\dfrac{\dfrac{\dfrac{\Gamma_{\texttt{nz}}, \texttt{t} : \texttt{int} \vdash s_1 \quad \Gamma_{\texttt{mod}} \vdash s_2 \quad \Gamma_{\texttt{mod}} \sqsubseteq \Gamma_{\texttt{nz}}, \texttt{t} : \texttt{int}}{\Gamma_{\texttt{mod}} \vdash s_1; s_2}}{\Gamma_{\texttt{mod}} \vdash \texttt{do}\{s_1; s_2\}} \quad \dfrac{\Gamma \vdash \texttt{asg} : \{\texttt{det}\}(\texttt{int}, ?\texttt{int})}{\Gamma, \texttt{x} : \texttt{int}, \texttt{z} : \texttt{int} \vdash \texttt{asg(x}, ?\texttt{z)}}}{\Gamma_{\texttt{mod}}, \texttt{z} : \texttt{int} \vdash \texttt{do}\{s_1; s_2\}; \texttt{asg(x}, ?\texttt{z)}}$$

(C) Type checking $s \coloneqq \texttt{do}\{s_1; s_2\}; \texttt{asg(x}, ?\texttt{z)}$ in the `gcd` procedure.

$$\dfrac{\Gamma, \texttt{x} : \texttt{int}, \texttt{t} : \texttt{int}, \texttt{y} : \texttt{int}, \texttt{z} : \texttt{int} \vdash s}{\Gamma \vdash \texttt{def gcd(x} : \texttt{int}, \texttt{y} : \texttt{int}, ?\texttt{z} : \texttt{int})\{s\}}$$

(D) Type checking of the definition of the `gcd` procedure.

FIGURE 4.7: Proof of type correctness of the `gcd` procedure.

Sequencing s_1 and s_2 in Figure 4.7C is well typed under $\Gamma_{\texttt{mod}}$. The sequence of $s = \texttt{do}\{s_1; s_2\}; asg(x, ?z)$ is similarly well typed under $\Gamma_{\texttt{mod}}, \texttt{z} : \texttt{int}$, using the well-typing of `do` block and the same instantiation of `asg`.

In result, we have shown that in the body of `gcd`, s is well-typed, with the typing of parameters, `x:int`, `y:int`, `z:int`. Hence, in Figure 4.7D we conclude the definition of `gcd` is type correct.

4.3 Wybe Type Checking Algorithm

With the rules defined in Section 4.2, we can construct an algorithm that performs both type checking and type inference. This algorithm checks for type-correctness of all Wybe modules in a program. We do not define the algorithm in detail due to the work not being part of the work of this thesis, however, we provide a sketch of the algorithm and the relevant extensions made to enable higher-order programming.

Prior to the algorithm's execution, terms present in the AST are flattened into a constrained subset of AST terms with equivalent semantics. For example, a function call expression, `f(x, y)` is transformed into a preceding statement, `f(x, y, ?`*tmp*`)`, and the fresh variable *tmp* is used in its place. Similarly, statements such as `f(x) = ?y` are transformed into `f(x, ?y)` statements. This simplifies the type inference algorithm, as fewer expressions are required to be handled in the algorithm and corresponding rules, due to these expressions being flattened into statements.

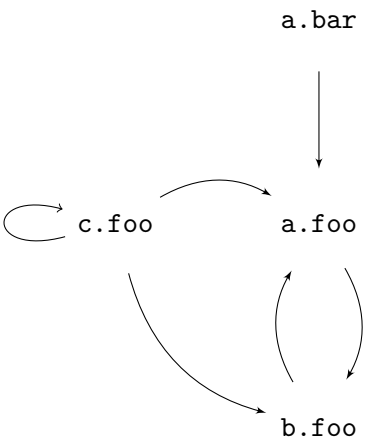
A point where this algorithm diverges from algorithms, such as Algorithm W [53], which implements a type inference algorithm using the rules defined by the HM type system, is the consideration of overloading. Overloading is not handled in such algorithms but must be considered in the Wybe language, due to the presence of overloading in the language. Overloading is implicit in the typing rules defined in Section 4.2, as where there may be multiple defined procedures with the same name, one is chosen non-deterministically in the type checking of a call. In the type checking algorithm, however, overloading must be handled explicitly.

With procedure overloading, a *potential call graph* arises. The potential call graph considers the possible overloading resolutions across all calls in the body of each procedure, with a node in the graph representing a procedure, and an edge corresponding to a *potential call*. That is, if foo calls some procedure *bar*, then there is a directed edge from foo to all procedures that resolve to *bar*. Module qualifications to the name allow for this set of procedures to be constrained, such that the qualified module is a super-module of the potential procedures' modules. The potential call graph does not consider the mode, nor arity of the calls, merely the (module qualified) name.

A concrete example of a translation from a set of Wybe modules to a potential call graph is found in Figure 4.8. For these modules, the SCCs in bottom-up order of the potential call graph are: {a.foo, b.foo}, {c.foo}, {a.bar}.

```
1  module a {
2      pub def foo(x) { b.foo(x) }
3      pub def bar(x) { a.foo(x) }
4  }
5
6  module b {
7      pub def foo(x) { a.foo(x) }
8  }
9
10 module c {
11     def foo(f, x) {
12         foo(x)
13         f(x)
14     }
15 }
```

(A) Wybe program without all explicit module qualifications.



(B) The potential call graph of the Wybe program.

FIGURE 4.8: Translation of a Wybe program into a potential call graph.

With the potential call graph, the algorithm considers the strongly connected components (SCCs) of the graph. The SCCs define an ordering on procedures, with callee procedures considered after the callers. As the semantics we desire for Wybe specify that the typing of a procedure depends only on the definition, not on the typing of any calling procedure, procedures must be considered bottom-up through the potential call graph.

Higher-order calls are also recognised here. A call to some non-module qualified name where the name is bound to a higher-order type is recognised as a higher-order call. In the type inference algorithm, we exclude all other overloadings for this call, enforcing that this call must be to the higher order term defined elsewhere in the procedure. An example of such a call that can be recognised as a higher-order call is the call to `f` in `c.foo` of Figure 4.8.

With this ordering, however, the algorithm must handle a cyclic SCC carefully. As the typing of a procedure depends on the procedures it calls, the typing of a called procedure may not fully be inferred. As such, the typing of each procedure is approximated to the

most general type, with the types of procedures being refined through each iteration of type checking over each procedure in the cyclic SCC. This process can terminate if there is some un-recoverable type error or a fixed point is reached where the typings cannot further be refined.

This algorithm is also performed in two parts: type checking and mode checking. Type checking is mode-agnostic and is implemented to discover the typing of all variables in a given procedure. After, mode checking is performed top-down on the AST, modifying procedure calls into their fully module-qualified selves. Mode checking further ensures that variables are correctly bound, requiring that a variable is unconditionally bound before it is used, and that resources are appropriately in scope for all calls.

In type checking a procedure's body, overloading can lead to multiple correct typings. In such a case, the algorithm reports an overloading error. In Listing 4.2, we see a program that defines two overloaded versions of `add2` implementing an integer and floating point sum of two numbers, respectively. In the definition of `add3`, two calls are made to `add2`. However, with no constraints on the types of the inputs, the overloading resolution leads to two possible typings, where all of `a`, `b`, `c`, and `d` have either an inferred type of `int` or `float`, and as such reports an overloading error. This error can be resolved with the use of a explicit type constraint on any of the variables in the body of `add3`.

```
1  def add2(a:float, b:float, ?c:float) {
2      # omitted implementation of floating point add
3  }
4
5  def add2(a:int, b:int, ?c:int) {
6      # omitted implementation of integer add
7  }
8
9  def add3(a, b, c, ?d) {
10     add2(a, b, ?t)
11     add2(t, c, ?d)
12 }
```

Listing 4.2: Example program where overloading cannot be resolved.

Note that the ordering of calls is irrelevant in the body of a type correct procedure and that only calls can be used to place constraints on variables. As such, the algorithm does not consider non-call statements, however does consider the nested calls in such statements. All non-call statements place constraints on the variables inside the body of

a procedure, and these constraints along with programmer-defined constraints are used to construct an approximate typing context. Each call is also paired with each possible resolution of such a call. The algorithm maintains the invariant that only possibly type correct overloading resolutions are kept.

The algorithm in place in the Wybe compiler performs a heuristic backtracking search. With a candidate typing context, type constraints can eliminate possible overloading resolutions. If no possible overloadings remain, the candidate typing is abandoned and a type error is reported. If multiple candidate typings are found to be correct for all calls, an overloading error is reported and all such candidate typings are abandoned.

Chapter 5

Transformations of Resources

This chapter introduces the transformations we require in transforming resources into lower-level constructs. The existing transformation in the Wybe compiler parameterises resoruces, transforming resources into formal parameters and arguments.

With the extended, higher-order, semantics we propose for the Wybe language in Section 3.2, we provide an implementation strategy that performs a second transformation pass, transforming resources into global variables. Finally, we discuss a limitation of the globalisation transformation and a potential resolution to the discussed limitation.

5.1 Parameterisation of Resources

While resources in the Wybe language are passed implicitly by name, the intermediate representation used within the Wybe compiler does not support a by-name parameter passing mechanism, providing only the usual positional parameter passing mechanism. As such, the Wybe compiler must transform a procedure and all resourceful procedure calls to follow a positional parameter passing scheme. To transform resources in a Wybe program into regular parameters, the compiler performs the translation outlined in Figure 5.1.

Throughout this transformation, we impose an ordering of resources. We define an ordering of resources by module first and then by name. This ordering is used to ensure that resources are consistently passed are arguments in the same order as the corresponding parameters.

In Figure 5.1 the R_{proc} rule transforms a procedure's resources into parameters, by appending the list of resources in canonical order to the list of parameters.

$$R_{proc}[\![\texttt{def}\ proc(params)\ \texttt{use}\ res\ \{b\}]\!] = \texttt{def}\ proc(params + res)\ \texttt{use}\ res\ \{R_{stmts}[\![b]\!]\}$$

$$\begin{aligned} R_{stmts}[\![[\ ]]\!] &= [\ ] \\ R_{stmts}[\![[stmt \mid stmts]]\!] &= R_{stmt}[\![stmt]\!] + R_{stmts}[\![stmts]\!] \end{aligned}$$

$$\begin{aligned} R_{stmt}[\![proc(args)]\!] &= [proc(args + resources(proc))] \\ R_{stmt}[\![\texttt{use}\ r_1, \ldots, r_n\ \texttt{in}\ b]\!] &= [move(r_i, ?tmp_i), \ldots, move(r_n, ?tmp_n)] \\ &\quad + R_{stmts}[\![b]\!] + \\ &\quad [move(tmp_1, ?r_1), \ldots, move(tmp_n, ?r_n)] \\ R_{stmt}[\![other]\!] &= R_{nested}[\![other]\!] \end{aligned}$$

FIGURE 5.1: Parameterisation of resources, transforming resources in a procedure into formal parameters and arguments.

A sequence of statements, represented by a Prolog-style list, $[stmt_1|[stmt_2|...]]$, and terminated by an empty sequence, [ ], transforms each statement in sequence. As each statement may be transformed into numerous statements, the resultant sequence of statements, $R_{stmt}[\![stmt]\!]$ is appended to the transformed result of the remaining statements.

Two key statements must be transformed to allow for procedures to pass resources explicitly. The first is a (resourceful) procedure call. Each procedure call is transformed by adding explicit arguments. These resource arguments follow the ordering defined for resources, ensuring that the positions of all resource arguments are consistent with the resource parameters as defined previously.

The other statement that must be transformed is the `use` block. Semantically, a `use` block must obey two properties: the value of each used resource must be identical directly before and after the `use` block, and any resource previously not in-scope is considered in-scope for the duration of the block. To ensure the first property, each resource that is bound immediately before the `use` block is assigned to a temporary variable. After the `use` block. These temporary variables are re-assigned to the resource variable. This ensures that the values of the resources are identical immediately before and after the block. The second property is irrelevant to this transformation due to the scope and binding state of resources being checked before this transformation.

All other statements do not require any special transformations, only requiring the transformation of all nested statements. For the sake of brevity in Figure 5.1, these transformations have been omitted, replaced by the $R_{stmt}[\![other]\!]$ transformation.

5.2 Resources as Global Variables

In this section, we detail the implementation strategy for the proposed higher-order resource semantics outlined in Section 3.2. As resources may be used by some procedure while simultaneously being used by some higher-order term, the compiler is no longer able to transform resources into arguments and parameters of higher-order calls. This is ultimately due to higher-order terms lacking an annotation of exactly which resources the term may use, but only some collection of unknown resources. As the compiler does not know which resources the terms may use, the compiler cannot pass the resources as positional arguments.

However, we can overcome this issue with a uniform name that can be used to access the current value of the resource. This name should be accessible in any given context, being shared across all contexts, be it in a higher-order call or a regular procedure body. This is exactly the interface a global variable provides: a single, uniform name, that is accessible in any context.

With global variables available in any context, global variables are less constrained than desired for resources. As such, in the transformation of resources from formal parameters and arguments to global variables is outlined in Section 5.2.1, we must constrain their usage to be aligned with that of the represented resources. This also ensures that the semantics of a globalised program is maintained.

To prevent unwarranted manipulation of these global variables from the programmer, this transformation is transparent for a programmer, with identical semantics for a first-order program. The programmer can still manipulate the resource as previously possible.

5.2.1 Globalisation of Resources

Minor changes must be made to the existing resource transformation as outlined in Section 5.1. Due to expressions now possibly containing statements in anonymous procedures, the transformation must also perform any transformations to those anonymous procedures as is performed for any regular procedure. `use` blocks were omitted after the prior transformation, however, due to their usage in the globalisation transformation, are retained after the transformation.

$$
\begin{aligned}
R_{stmt}[\![proc(args)]\!] &= [proc(R_{nested}[\![args]\!] + resources(proc))] \\
R_{stmt}[\![\texttt{use } r_1, \ldots, r_n \texttt{ in } b]\!] &= [move(r_i, ?tmp_i), \ldots, move(r_n, ?tmp_n)] \\
&\quad + [\texttt{use } r_1, \ldots, r_n \texttt{ in } R_{stmts}[\![b]\!]] \\
&\quad + [move(tmp_1, ?r_1), \ldots, move(tmp_n, ?r_n)]
\end{aligned}
$$

FIGURE 5.2: Modifications to the transformation of resources into parameterised resources.

We are unable to omit the transformation pass for parameterised resources due to other passes within the compiler being present that rely on the parameterisation transformation pass being performed, such as a pass that checks for the validity of linear types [20] in a Wybe program. With the modified transformations of resources in place and other passes being performed appropriately, the compiler is now able to transform a Wybe program with parameterised resources into a Wybe program with globalised resources, as outlined in Figure 5.3. We denote a reference to a global variable g as the term $\langle g \rangle$.

To transform a procedure, the transformation begins by removing the parameters that correspond to resources as introduced by the parameterisation transformation. The procedure's body, a list of statements, is also transformed noting the set of resources, res, that are currently in scope for the procedure body. In this transformation, it is assumed that resources are appropriately in-scope and bound, a property that is achieved after type analysis. To ensure that the value of an in-flowing resource is not modified after a call to the procedure, an additional `use` block is introduced for each in-flowing resource, ensuring that the value is unchanged due to the transformation of `use` blocks .

There are two parts to transforming a procedure call. First, each argument that corresponds to a resource argument, as was introduced in the resource parameterisation

$$
\begin{aligned}
G_{proc}[\![\texttt{def}\ proc(params)\ \texttt{use}\ res\ \{b\}]\!] &= \texttt{def}\ proc(params - res)\ \texttt{use}\ res\ \{b'\} \\
&\text{where } ins = \{r \in res \mid flow(r) = in\} \\
&\quad b' = G^{res,\varnothing}_{stmts}[\![\texttt{use}\ ins\ \texttt{in}\ b]\!]
\end{aligned}
$$

$$
\begin{aligned}
G^{r,t}_{stmts}[\![[\,]]\!] &= [\,] \\
G^{r,t}_{stmts}[\![[stmt \mid stmts]]\!] &= G^{r,t}_{stmt}[\![stmt]\!] + G^{r,t}_{stmts}[\![stmts]\!]
\end{aligned}
$$

$$
\begin{aligned}
G^{r,t}_{stmt}[\![proc(args)]\!] &= loads + [proc(args')] + stores \\
&\text{where } (args', loads, stores) = G^{r,t}_{args}[\![args - resources(proc)]\!] \\
G^{r,t}_{stmt}[\![\texttt{if}\{cond :: thn \mid \texttt{else} :: els\}]\!] &= [\texttt{load}(\langle r_i\rangle, ?tmp_i) \mid r_i \in r] \\
&\quad + [\texttt{if}\ \{\ G^{r,t}_{stmt}[\![cond]\!] :: G^{r,t}_{stmts}[\![thn]\!] \\
&\qquad \mid \texttt{else} :: [\texttt{store}(tmp_i, \langle r_i\rangle) \mid r_i \in r] \\
&\qquad\quad + G^{r,t}_{stmts}[\![els]\!]\}] \\
G^{r,t}_{stmt}[\![\texttt{use}\ r_1, \ldots, r_n\ \texttt{in}\ b]\!] &= [\texttt{load}(\langle r_1\rangle, ?tmp_1), \ldots, \texttt{load}(\langle r_n\rangle, ?tmp_n)] \\
&\quad + G^{r\cup\{r_1,\ldots,r_n\},\ t\cup\{r_i\mapsto tmp_i \mid r_i\notin t\}}_{stmts}[\![b]\!] \\
&\quad + [\texttt{store}(\langle r_1\rangle, tmp_1), \ldots, \texttt{store}(\langle r_n\rangle, tmp_n)] \\
G^{r,t}_{stmt}[\![\texttt{do}\{body\}]\!] &= G^{r,\varnothing}_{stmts}[\![body]\!] \\
G^{r,t}_{stmt}[\![\texttt{break}]\!] &= [\texttt{store}(t[g], \langle g\rangle) \mid g \in t] + [\texttt{break}] \\
G^{r,t}_{stmt}[\![\texttt{next}]\!] &= [\texttt{store}(t[g], \langle g\rangle) \mid g \in t] + [\texttt{next}] \\
G^{r,t}_{stmt}[\![other]\!] &= G^{r,t}_{nested}[\![other]\!]
\end{aligned}
$$

$$
G^{r,t}_{args}[\![args]\!] = (G^{r,t}_{nested}[\![args]\!], G^{r}_{load}[\![args]\!], G^{r}_{store}[\![args]\!])
$$

$$
\begin{aligned}
G^{r}_{load}[\![v]\!] &= [\texttt{load}(\langle v\rangle, ?v) \mid v \in args,\ v \in r,\ flow(v) \neq out] \\
G^{r}_{store}[\![v]\!] &= [\texttt{store}(v, \langle v\rangle) \mid v \in args,\ v \in r,\ flow(v) \neq in]
\end{aligned}
$$

FIGURE 5.3: Globalisation of resources, transforming parameterised resources into global variables.

transformation (Section 5.1), is removed. These arguments are no longer necessary to follow the new interface to which this transformation conforms.

Following the removal of resource arguments, the non-resource arguments are transformed with the G^{r}_{args} transformation. Each argument that is a variable may introduce an additional statement to either `load` or `store` some resource. If the variable is a resource variable for some resource $v \in r$ that is currently in scope, such instructions are introduced depending on the flow of the argument. Where some argument flows in (as an input or in/output, *i.e.*, not an output), the value of the global variable must be

loaded before (G^r_{load}). If the flow of the argument flows out (as an output or in/output, *i.e.*, not an input), then the argument must be stored after the procedure call (G^r_{store}). Each `load` and `store` instruction loads the global variable into the variable v or stores the value of v into the global variable, respectively.

If some procedure that modifies some resource fails, the value of the resource should be unmodified if the procedure fails, as occurs with parameterised resources. This applies to conditional statements also. If the condition of some conditional statement fails, then the value of the resource must be restored to the value before the condition is executed before continuing with the else branch.

The transformation achieves this by, for each in-scope resource, $r_i \in r$, saving the value of the resource's global variable in some fresh variable, tmp_i, using a $\texttt{load}(\langle r_i \rangle, ?tmp_i)$ instruction. These instructions are inserted before each conditional statement. The fresh variables, tmp_i, are then used to restore the value of each global variable prior to the statements in the else branch of the conditional statement with a prepended $\texttt{store}(tmp_i, \langle r_i \rangle)$ instruction for each r_i. These instructions are only necessary where the condition may modify some resource, in which case `load` and `store` instructions are only necessary for resources that may be overwritten in the condition.

For each resource marked in the `use` block, r_i, a $\texttt{load}(\langle r_i \rangle, ?tmp_i)$ instruction stores the current value of the global variable into a fresh variable, tmp_i. Each now in scope resource is added to the set of in-scope resources in the transformation of the nested statements of the `use` block. After the transformed nested statements, a $\texttt{store}(tmp_i, \langle r_i \rangle)$ instruction is added for each resource, r_i in the `use` block, restoring the value of the global variable to the value prior to the `use` block. In performing this transformation, the value of resources in the global variables immediately before and after the `use` blocks remain unchanged, even if the value of some global variable is trampled inside the `use` block.

The `break` and `next` instructions are able to modify the course of execution of a procedure, jumping to the end and start of a `do` loop, respectively. It is possible for these instructions to be contained within a `use` block that is inside their respective `do` loop. When this occurs, the value of a resource's global variable is required to be restored to that of before the `use` block. This property is achieved through the insertion of $\texttt{store}(t[g], \langle g \rangle)$ instructions for each resource, g, before all `break` and `next` instructions.

Only the resources that have been previously saved from a `load` instruction inside the loop, and only the outermost such fresh variable, $t[g]$, is required to be restored. The current values of the resources (and corresponding fresh variables) to be restored are stored in the t map. The map, t, is emptied in the recursive transformation of a `do` loop's body, and added to for each `use` block, adding only temporary variables for each resource $r_i \notin t$. This ensures that only the outermost temporary variable is restored upon a `break` or `next` statement, as this is the value that must be restored.

All other statements and any nested expressions also must be transformed accordingly, transforming nested statements and variables. These are obfuscated in Figure 5.3 with the $G^r_{stmt}[\![other]\!]$ and G^r_{nested} rules for the sake of brevity.

```
1  resource counter:int
2
3  def tick() use !counter {
4      ?counter = counter + 1
5  }
6
7  def tick_print() use counter, !io {
8      !tick()
9      !print(counter)
10 }
```

(A) Before parameterisation.

```
1  resource counter:int
2
3  def tick(!counter) {
4
5      ?counter = counter + 1
6
7  }
8
9  def tick_print(counter, !io) {
10
11     !tick(!counter)
12
13     !print(counter, !io)
14
15 }
```

(B) After parameterisation.

```
resource counter:int

def tick {
    load(<counter>, ?counter)
    ?counter = counter + 1
    store(counter, <counter>)
}

def tick_print {
    load(<counter>, ?tmp1)
    !tick()
    load(<counter>, ?counter)
    !print(counter)
    store(tmp1, <counter>)
}
```

(C) After globalisation.

LISTING 5.1: Example of the transformation progression of resources. The changes made after each transformation are highlighted in red.

An example of the progression of resources through the various transformations is provided in Listing 5.1, exemplifying the transformation of resources into formal parameters and arguments, and finally into global variables.

5.2.2 Limitations of Globalisation

We are aware of one limitation of the implementation strategy of transforming resources into global variables. We exemplify this limitation with the use of an example in Listing 5.2, where the globalisation fails to adhere to the desired semantics.

```
1  resource res:int = 0
2
3  def foo(f:{resource}()) {
4      if { !bar(f) :: pass }
5  }
6
7  def {test} bar(f:{resource}()) {
8      !f()
9      fail
10 }
11
12 ?f = {resource}{ ?res = 1 }
13
14 # before:   res = 0
15 !foo(f)
16 # expected: res = 0
17 # after:    res = 1
```

LISTING 5.2: Example where the globalisation implementation strategy invalidates the desired semantics.

In Listing 5.2, the term `f` uses the `res` resource, modifying the value of the resource to be `1`. The procedure `foo` is called with `f` as an argument. Prior to this call, the `res` resource is initialised to the value of `0`. Inside the call to `foo`, `f` is passed as an argument to the `bar` procedure.

The call to `bar` should only propagate changes to resources if the condition succeeds. However, investigating the implementation of `bar`, we see that the procedure must always fail (`fail`). This failure occurs after the higher-order call to `f` is performed, possibly modifying some resource.

The call to `f` does modify the resource, as the argument to this procedure re-assigns the resource to `1`. As `bar` fails, the value of any resource modified in `bar` should be restored back to their prior values. However, due to the semantics we defined for resources in a higher-order context, we cannot be certain which resources must be restored. One might imagine that the `bar` procedure is defined in some distinct module that has no knowledge that the `res` resource exists. As the value of `res` does not get restored to the value prior to the call to `bar`, this change propagates outside the call to `foo`. In effect, this means that the value of `res` is now `1` after the call to `foo`.

In Prolog and other logic programming languages, the standard solution to backtracking state changes on failure is using a `trail` [54]. A trail is a separate stack that is used to track state changes of variables that are to be reset on failure. Upon failure, the trail is traversed backwards, restoring the state of any variables before the failing call. In future work, we may be able to resolve this limitation of globalisation with the use of a similar structure. This would require saving the state of global variables before entering a non-deterministic call, and restoring the state in the case that the call fails.

Chapter 6

Translation to LPVM and Extensions of LPVM

This chapter details the transformation process from a Wybe AST program into the LPVM intermediate representation. The transformation occurs in two parts, being an *unbranching* pass that constrains the AST into a form that can then be translated into LPVM following a variable numbering pass. While these processes are largely unchanged, we introduce the extensions we require for higher-order programming in detail.

We also introduce the LPVM intermediate representation, introducing the extensions to the language required for higher-order programming and global variables.

6.1 Unbranching

Unbranching is a transformation of a Wybe procedure into a set of mutually recursive, deterministic, procedures. These procedures are constrained to closely align to the structure of LPVM, allowing for ease of translation from a transformed procedure into LPVM.

The procedures produced by the transformation have a simple form, being composed of a list of deterministic procedure calls, terminated by an optional conditional statement.

The branches of this conditional statement have the same form, with further optional terminating conditional statements.

This transformation is performed after resources have been transformed into their final representation, now as global variables. This ensures that constructs such as `use` blocks have been replaced by constructs that can be readily translated directly into LPVM. Of the constructs that do remain, some cannot readily be translated into LPVM, namely non-deterministic calls, non-tail conditional statements, and loops. We do not present this transformation formally as the transformation is largely unchanged, however, the extensions required with higher-order are important regarding future transformations.

The transformation produces a series of additional procedures, called *continuations*. These continuations differ from the continuations found in CPS [55], where in CPS they are higher-order calls, however, here they also occur at the tail position. Continuations are created after each `test` procedure call, non-tailing conditional statement, and loops. Continuations contain the statements that are to be performed after each such construct. Each continuation is naïvely constructed with inputs being all in-scope variables and all outwards-flowing parameters.

Unbranching a sequence of statements is performed in reverse. In reverse, the transformation maintains a list of already unbranched statements and a list of alternative statements that are to be executed in the case that some condition fails. These alternative statements are initially an assignment of `false` to a fresh variable in the case the unbranched procedure is a `test` procedure and empty otherwise. This fresh variable is added to the end of the list of formal parameters as an output, de-reifying the procedure into a `det` procedure.

Calls, both first-order and higher-order are unbranched similarly. In the case of a `test` call determinism, we transform this statement into a de-reified `det` call and tailing conditional branch. The statement is augmented with an additional outwards-flowing Boolean-typed argument, which is then used as the test in the condition, with the *then* and *else* branches containing the list of unbranched and alternative statements, respectively.

Both `terminal` and `failing` calls are unbranched similarly, with the following unbranched being discarded. In the case of a `failing` procedure, however, the alternative

```
1  def count_down(x) {
2      do {
3
4
5          # sugared test
6          if { x > 0 ::
7
8              break
9          | else ::
10             println(x)
11             ?x = x - 1
12
13         }
14     }
15
16
17     println("done")
18 }
```

(A) Before unbranching.

```
def count_down(x) {
    next(x)
}

def next(x) {
    `>`(x, 0, ?tmp) # de-sugared
    if { tmp ::
        break(x)
    | else ::
        println(x)
        ?x = x - 1
        next(x)
    }
}

def break(x) {
    println("done")
}
```

(B) After unbranching.

LISTING 6.1: Example of an unbranched procedure. Type annotations are omitted for the sake of brevity, and additional procedure calls and procedure definitions are highlighted in red.

statements are placed after the call. `det` calls are otherwise unchanged, with the list of unbranched statements following.

Loops are unbranched by producing two fresh continuation procedures, *next* and *break*, as seen in Listing 6.1. The *next* procedure is a continuation of all statements contained within the loops. The *break* and *next* procedures are used to create mutually recursive procedures that are equivalent, semantically, to the loop and following statements.

The *next* procedure is called in place of the loop as a tail call in the unbranched procedure and replaces `next` statements within the loop itself. The *break* procedure is a continuation of the statements that occur after the loop. The `break` statements inside the loop body are replaced with a tail call to the *break* procedure.

After a procedure has been unbranched, the constraints placed on the AST ensure that the structure is identical to that of LPVM (see Section 6.3). As such, the translation to LPVM is simplified, requiring only that each variable reassignment introduces a new version of the variable, much like the versions of variables used with ϕ-nodes in SSA form [38]. The acyclic nature of LPVM, however, precludes the requirement of such

nodes in the intermediate representation. An example of this version numbering is seen in Listing 6.2.

```
1  def count_down(x0) {
2      next(x0)
3  }
4
5  def next(x0) {
6      `>`(x0, 0, ?tmp0) # de-sugared
7      if { tmp0 ::
8          break(x0)
9      | else ::
10         println(x0)
11         ?x1 = x0 - 1
12         next(x1)
13     }
14 }
15
16 def break(x0) {
17     println("done")
18 }
```

LISTING 6.2: Procedure with re-assigned variables as new versions of the variable with an affixed number. Note that once re-assigned, the variable is never used again. This example continues from Listing 6.1.

6.2 Closure Conversion

With the advent of higher-order code in Wybe, expressions now may contain two higher-order constructs, namely anonymous procedures, and partial applications. For simplicity, a uniform representation of these constructs allows for ease of transformation in latter optimisations (Chapter 7) and translations (Chapter 8). We perform a variant of closure conversion [10] in two stages, first by hoisting anonymous procedures into regular procedures, then generating trampolines to represent closures for all procedure references.

All anonymous procedures are converted into regular procedures. This hoists the anonymous procedure into the module level, generating a fresh procedure, with all holes by obfuscated variable names. This procedure is also unbranched as with all other procedures.

The parameters of this hoisted procedure can be categorised into two groups, free variables and holes, with the free variables present in the parameter list before the holes. Each hole is given a name uniformly throughout the body of the procedure. Free variables are recognised here as any variable that is currently in scope and also used within the body of the hoisted procedure.

The anonymous procedure expression is then replaced with a reference to the hoisted procedure, as a partial application of the now-closed variables. As these variables are capured by value, not by reference, passing the value to the partial application enables the desired behaviour. An example of this can be seen in Listing 6.3.

```
1  def foo(b:bool,
2          ?f:(int, int, ?int)) {
3      ?f = {
4
5
6
7          if { b ::
8              @1 = ?@3
9          | else ::
10             @2 = ?@3
11         }
12     }
13 }
```

(A) Before hoisting.

```
def foo(b:bool,
        ?f:(int, int, ?int)) {
    ?f = anon<b>
}
def anon(^b:bool,
         p1:int, p2:int, ?p3:int) {
    if { b ::
        p1 = ?p3
    | else ::
        p2 = ?p3
    }
}
```

(B) After hoisting.

LISTING 6.3: Example of the first stage of closure conversion, hoisting. The anonymous procedure is converted into a fresh procedure, `anon`, with `b` as a free variable (marked by a preceding `^`) and renamed holes, with the relevant modifications are highlighted in red.

The second stage of closure conversion converts all procedure references (including those to hoisted anonymous procedures) into a reference to a closure procedure. After this conversion is performed, all procedure references are now references to closure procedures.

A closure procedure for some procedure, p, with closed variables, C, is simple. The procedure has an identical interface to p and contains a single call to p inside the body, with closed variables present before the regular arguments. However, the parameters that correspond to the closed variables, C, are marked as being free variables. An example of this process is found in Listing 6.4.

```
1  def foo(b:bool, ?f:(int, int, ?int)) {
2      ?f = clos<b>
3  }
4  def anon(^b:bool, p1:int, p2:int, ?p3:int) {
5      if { b ::
6          p1 = ?p3
7      | else ::
8          p2 = ?p3
9      }
10 }
11 def clos(^b:bool, p1:int, p2:int, ?p3:int) {
12     anon(b, p1, p2, ?p3)
13 }
```

LISTING 6.4: Example of the second stage of closure conversion in unbranching, continuing from Listing 6.3. The procedure reference, `anon<b>`, is transformed into a fresh closure procedure, `clos`, with changes highlighted in red.

The ultimate purpose of a closure procedure is in the final stages of the Wybe compiler, in translation to LLVM (Chapter 8), creating a closure with a reference to its environment. In the final stages of the compiler, in translation to LLVM (Section 8.2), these closure procedures are transformed into a conventional closure with an explicit *environment* parameter which is used to retrieve the values of closed variables. We delay this transformation to allow greater use of optimisations of the generated LLVM code, namely neededness analysis on the closed variables (see Section 7.1.1) while ensuring the interface of this procedure is conserved.

6.3 Extended LPVM Implementation

Logic Programming Virtual Machine (LPVM) [41] is an LP form intermediate representation based upon constrained sets of horn clauses as a program representation. LPVM is the primary intermediate representation in place in the Wybe compiler.

The form of LPVM presented and extended here and used within the compiler differs from initially presented in [41], following more closely to the outlined tree-based implementation. This implementation of LPVM also allows for impurity, while the original presentation of the representation was intended to be pure. Impurity inhibits some optimisations that could occur with LPVM, and as such procedures are assumed to be pure unless otherwise annotated.

In LPVM, procedures have a simple form, as defined by the grammar in Figure 6.1. Procedures have a list of formal parameters and accompanying flows, with flows restricted to being either inwards or outwards (`?`) (with in/out arguments replaced by a separate inwards and outwards flowing argument). All procedure bodies are a (possibly empty) list of *primitives*, optionally terminated by a *fork*. A fork is a conditional branch, being conditioned upon some variable with several cases. Each case corresponds to a value, such that if the variable has said value, the body of the case is executed.

$$
\begin{array}{rcl}
\langle \text{proc} \rangle & ::= & \mathit{name}\texttt{(}(\langle \text{flow} \rangle \mathit{var})^*\texttt{):}\ \langle \text{body} \rangle \\
\langle \text{body} \rangle & ::= & \langle \text{prim} \rangle^*\ \langle \text{fork} \rangle \\
\langle \text{prim} \rangle & ::= & \mathit{name}\texttt{(}\langle \text{val} \rangle^*\texttt{)} \\
 & | & \texttt{foreign}\ \mathit{lang}\ \mathit{name}\texttt{(}\langle \text{val} \rangle^*\texttt{)} \\
\langle \text{fork} \rangle & ::= & \texttt{case}\ \mathit{var}\texttt{:}\ \langle \text{case} \rangle^* \ \mid\ \epsilon \\
\langle \text{case} \rangle & ::= & \mathit{const}\texttt{:}\ \langle \text{body} \rangle \\
\langle \text{val} \rangle & ::= & \langle \text{flow} \rangle \mathit{var} \ \mid\ \mathit{const} \\
\langle \text{flow} \rangle & ::= & \texttt{?} \ \mid\ \epsilon
\end{array}
$$

FIGURE 6.1: The grammar of the LPVM intermediate representation.

Primitives are the procedure calls of LPVM. There are two variants of primitives, foreign and regular calls. Foreign calls provide the foreign function interface in LPVM, with a specified language, such as LLVM which is used to interface with low-level instructions including those for arithmetic and Boolean operations. Foreign calls also include the built-in procedures of the LPVM language, handling elements of the language such as memory management and assignment via the `move` instructions. Regular calls are all other calls, calling procedures defined in some LPVM module.

With the advent of higher-order code, there are two features that we require in the LPVM implementation, namely higher-order calls and terms, and global variables and global flows. The extended grammar of LPVM is presented in Figure 6.2.

We extend the concept of a primitive to include higher-order calls. Similar to a first-order primitive call, we make a higher-order call with a list of arguments. However, unlike a first-order primitive call, we do not make the call to a static name, but to some value. For a program in LPVM to be semantically correct, the value called in a higher-order type must have the correct arity, flows, and types. These higher-order types are identical to those introduced in Chapter 4, with accompanying modifiers controlling the

purity and ability to manipulate resources (now global variables), however in/out flows are translated into a pair of in and out flows.

$$
\begin{array}{rcl}
\langle \text{proc} \rangle & ::= & \mathit{name}\texttt{(}(\langle \text{flow} \rangle \mathit{var})^*\texttt{)<}\langle \text{globals} \rangle\texttt{,}\langle \text{globals} \rangle\texttt{>:}\ \langle \text{body} \rangle \\
\langle \text{body} \rangle & ::= & \langle \text{prim} \rangle^*\ \langle \text{fork} \rangle \\
\langle \text{prim} \rangle & ::= & \mathit{name}\texttt{(}\mathit{val}^*\texttt{)<}\langle \text{globals} \rangle\texttt{,}\langle \text{globals} \rangle\texttt{>} \\
& | & \langle \text{val} \rangle\texttt{(}\langle \text{val} \rangle^*\texttt{)} \\
& | & \texttt{foreign}\ \mathit{lang}\ \mathit{name}\texttt{(}\mathit{val}^*\texttt{)} \\
\langle \text{fork} \rangle & ::= & \texttt{case}\ \mathit{var}\texttt{:}\ \langle \text{case} \rangle^*\ \mid\ \epsilon \\
\langle \text{case} \rangle & ::= & \mathit{const}\texttt{:}\ \langle \text{body} \rangle \\
\langle \text{val} \rangle & ::= & \langle \text{flow} \rangle \mathit{var}\ \mid\ \mathit{const}\ \mid\ \mathit{global}\ \mid\ \mathit{name}\texttt{<}\langle \text{val} \rangle^*\texttt{>} \\
\langle \text{flow} \rangle & ::= & \texttt{?}\ \mid\ \epsilon \\
\langle \text{globals} \rangle & ::= & \{\mathit{global}^*\}\ \mid\ \mathbb{U}
\end{array}
$$

FIGURE 6.2: The grammar of the extended LPVM intermediate representation. We omit types from this grammar, however, all values are explicitly typed.

We also extend LPVM with two new values, partial applications of (closure) procedures, and *global* variable references. Partial applications follow directly from the representation in the Wybe AST, being a reference to a defined *closure* procedure, with the closure of the free variables captured in an argument list. As with all values in LPVM, all values are explicitly typed, with these types translating directly from the Wybe types.

In support of global variables, we define two LPVM primitives (Listing 6.5) that translate directly from the Wybe AST counterparts used in Section 5.2.1. These instructions are introduced only in the resource globalisation pass and are not allowed to originate from the source code.

```
1 foreign lpvm load(<global>:type, ?var:type)
2 foreign lpvm store(val:type, <global>:type)
```

LISTING 6.5: LPVM primitive instructions for the manipulation of global variables.

Globals are references to global variables, denoted by $\langle g \rangle$, each corresponding to a resource in the Wybe program. LPVM programs that are semantically correct have global variable references in two locations, in LPVM `load` and `store` instructions. Global variables have the same type in all references to the global variable. Global variables are introduced by the compiler in the resource globalisaton transformation (Section 5.2.1),

being present in only `load` and `store` instructions. As global variables cannot be explicitly referenced in Wybe source code, global variables are also wholly controlled by the compiler via these instructions.

Semantically, the LPVM `load` instruction loads the current value of a global variable, binding an output variable to this value. The LPVM `store` instruction, conversely, stores a value, be it an in-flowing variable or some constant, as the current value for a global variable.

6.3.1 Global Flows

To facilitate the optimisation of the manipulation of global variables, knowledge of how global variables are manipulated by a primitive in LPVM is useful. To this effect, we define *global flows*. Global flows provide a declarative interface via which we can analyse the state of global variables in an LPVM program. The optimisations defined in Chapter 7 make heavy use of global flows and the global flow interface of procedures to safely perform the defined optimisations.

Global flows are, semantically, flows associated with global variables. An inwards global flow dictates a global variable being read via some LPVM `load` instruction. Conversely, an outwards global flow represents a global variable that is written to via an LPVM `store` instruction.

We represent global flows as a pair of sets of global variables, $<in, out>$, with the first component, *in*, containing the inwards flows and the second, *out*, containing the outwards flows. These sets can be universal, $\mathbb{U}$, representing global flows (in or out) of *all* global variables.

For each procedure, we define a *global flow interface*, which is represented identically to global flows, as a pair of sets of global variables. The interface states which global variables a procedure potentially manipulates, *i.e.*, if a global variable has an inwards or outwards flow in some procedure's interface, this procedure potentially reads from or writes to this global variable, respectively. In the case that a global variable has an outwards flow, but no corresponding inwards flow, this global variable is guaranteed to be overwritten. For inwards-only flows of global variables, the state of the global

variable is guaranteed to be the same before and after the call, however, it may be read and written to.

The global flows in the interface of a procedure are derived from the parameter types and resources used (and corresponding flows) of the procedure. If some parameter to the procedure has a higher-order type with the `resource` modifier, then we define the global flows of this procedure as having universal global flows, $<\mathbb{U}, \mathbb{U}>$. As a higher-order term can potentially cause *any* resource to be read from or written to, this procedure can potentially read from or write to any given resource. Otherwise, for each resource, r, with some given flow, the procedure has an inwards or outwards global flow dependent upon the flow of r in the procedure.

These sets may not truly represent the global flows that occur within a procedure, however. For instance, if a global variable is written to before it is ever read from, then the interface does not require the inwards global flow, which can occur when a resource has an outwards flow, and must be bound to a value before it is used. Further, if a global variable's value is the same before and after the procedure were called, the interface does not require the outwards global flow. This is ensured by the compiler inserting a `use` block, saving the initial value of inwards-only flowing resources, and restoring them at the end of the procedure, ensuring the value remains unchanged. These flows are in effect is transparent to the callee of a procedure and is not included in the interface. These guarantees are made by the callee save convention guaranteed by the resource globalisation transformation (Section 5.2).

The remaining primitives in LPVM also have corresponding global flows. The LPVM `load` and `store` instructions are defined as having $<\{g\}, \varnothing>$ and $<\varnothing, \{g\}>$ global flows, respectively, when manipulating a global variable g. All other foreign primitives are defined to have no global flows, $<\varnothing, \varnothing>$.

Higher-order primitives are defined as having no global flows, $<\varnothing, \varnothing>$, when the type of the called higher-order term does not have the `resource` modifier. Otherwise, if the `resource` modifier is present the higher-order primitive is defined to have $<\mathbb{U}, \mathbb{U}>$ global flows. This is because a resourceful higher-order term can potentially modify *any* resource, and as such must conservatively assume that all global variables may be read or written.

Chapter 7

LPVM Optimisations

This chapter introduces various optimisations to the LPVM intermediate representation we make use of in the Wybe compiler. The optimisations relate primarily to two language features that have been introduced in this thesis: higher-order terms and the manipulation of global variables.

Throughout earlier stages of the compilation process, including mode checking, resource transformations and unbranching, we are careful to introduce additional statements and constructs conservatively. We, however, are not overly concerned with producing optimal code in these stages. The purpose of these optimisations is to take existing LPVM code and produce a more optimised version of this code that produces the same effects as the existing code.

The optimisation passes we detail in this chapter are built upon the *body builder* framework. The body builder is a general framework that can be used to perform various optimisation passes over a given procedure body in LPVM. Body building is performed in two passes, forwards then backwards, traversing the body in the direction of execution and in reverse, respectively. The compiler uses the body builder framework to perform various optimisations, including procedure call inlining and multiple specialisations [3].

7.1 Extension of LPVM Optimisations

The body builder framework has many built-in optimisation passes that are performed in tandem with each other. In this section we introduce and extend three optimisations, namely neededness, value propagation, and common sub-expression elimination, to incorporate higher-order calls and terms. We also introduce an optimisation pass to transform some higher-order calls into first-order calls.

7.1.1 Neededness

Neededness is an analysis that was designed for the LP form intermediate languages [41]. Neededness extends the dead code elimination transformation, a transformation that removes redundant code.

In general, a variable x is needed if x is an output of the procedure, or x is used in some computation that produces some other value, y, that is needed. Due to the single assignment property of LPVM, each variable is defined exactly once in a clause. As assignment has no effects, variables that are assigned but are never needed can be removed. If a procedure call has no outputs that are needed, then we can also remove said procedure call if it is pure. This is because the call will have no side effects and nothing produced by the procedure call is ever used later.

With the advent of global variables, a pure primitive may have outputs beyond producing regular output variables. These outputs are in the form of outwards global flows. If a primitive has an outwards global flow for some global variable, g, then the primitive *may* overwrite the value of g. As this effect may be necessary for the remaining computation, if a primitive has some outwards global flow, this primitive is considered *needed* and cannot be omitted. We make this extension to neededness analysis concrete in Section 7.2.2.

Neededness analysis can find that some input parameters of procedures are unused, and hence passing such arguments is redundant. When performing neededness analysis, if some parameter to a called procedure is marked as un-needed, the corresponding argument can be replaced with a dummy value, reducing the uses of said argument.

In the translation to LLVM, parameters marked as un-needed are not translated into parameters in the corresponding LLVM function. We introduce one caveat to this rule.

Closure procedures, as introduced in Section 6.2, a procedure is marked as being used as the trampoline for some closure. This procedure has a particular interface with certain parameters corresponding to now closed variables. The other parameters define the interface assumed to be used for the trampoline and must be as such for correct LLVM code generation. Therefore, the non-closed variable parameters as such are always marked as needed.

In general, neededness can be performed over the whole program until a fixed point is found, wherein no more parameters can be marked as unneeded, and no more dead code can be eliminated, finding a fixed point of the neededness of parameters. We perform a reduced neededness analysis in the Wybe compiler that is performed in the backwards pass of a body builder. This reduced neededness analysis is performed per procedure, removing instructions marked as unneeded, and accordingly the parameters that are also marked as unneeded.

7.1.2 Value Propagation

Value propagation is a generalisation of the constant propagation optimisation. Constant propagation transforms variables that are a known constant value into said constant value. This optimisation alone is not very powerful but is used in tandem with other optimisations such as common sub-expression elimination and neededness to provide stronger optimisations.

Value propagation extends the propagation of constant values to that of variables. If some variable, v, is assigned via a call `move`$(x, ?v)$, the value of v can be replaced by x throughout the whole body of the procedure. Here, x can be any value, be it some constant or some other variable.

When used in tandem with neededness analysis, assuming that v is not an output of the procedure, this `move` instruction is now dead and can be removed. The instruction is dead because the output value, v, is no longer used in the body of the procedure.

In the context of higher-order calls, the called higher-order term may be replaced by some other value, a procedure reference, which is useful in performing the call lowering optimisation, lowering a higher-order call to a known procedure into a first-order call.

A procedure reference may also have nested terms, the closed variables, which may be replaced through value propagation.

7.1.3 Common Sub-Expression Elimination

In a common sub-expression elimination (CSE) optimisation, identical expressions are replaced by a temporary value that stores the evaluation of the expression [56].

In LPVM, CSE takes a slightly different form. Whereas some languages and intermediate representations have expressions, LPVM does not, instead only having primitives, be it procedure calls, foreign calls, and now higher-order calls. These primitives may also have numerous output values. The notion of a common sub-expression is replaced by a common primitive. A common primitive is a pure primitive that identical to some other primitive, excluding the outputs of said primitives.

If it is found that a pure primitive has been called previously in some procedure's body with identical inputs, the primitive can be replaced by `move` instructions, replacing each output, v, with the corresponding output, u, from the previously computed, identical, primitive, with a `move`$(u, ?v)$ instruction.

As the primitive is pure, the only effect the primitive can have is producing the outputs. The outputs of a pure procedure will always be identical, and as such replacing the outputs with those of an identical primitive is identical, semantically. We also omit this optimisation from primitives that have any global flows, as we cannot be certain that the state of the global variables is identical.

In conjunction with value propagation, this optimisation allows the outputs of the common primitive to be removed from the procedure entirely, assuming they are unused by not being an output of the procedure.

We extend this optimisation to include higher-order calls also. Much like first-order calls, a higher-order call to a pure term has no side effects, producing only the outputs for the primitive. As such, if the body builder finds that a call to an identical higher-order term is made with identical input arguments, the optimisation can replace the higher-order call primitive with a series of `move`$(u, ?v)$ instructions, for each output v and corresponding output v from the previous higher-order call primitive.

7.1.4 Higher-Order Call Lowering

A higher-order call, by its very nature, has associated overheads. These overheads are introduced by making the higher-order call itself and through the creation of the closure and its referenced environment. Through various optimisations, primarily value propagation, it is possible for a higher-order call to be made to a procedure reference directly. This optimisation is common in compilers that use higher-order programming [19], though the format of this optimisation differs slightly with the representation of closures and partial applications of LPVM.

In this case, the higher-order call has the form $proc{<}c_1, \ldots, c_n{>}(arg_1, \ldots, arg_m)$, where c_i are the closed variables for the procedure reference. After the closure conversion process in the unbranching stage of the compiler (Section 6.2), all procedure references are guaranteed to be references to closure procedures. These closure procedures are defined to ensure that the interface of the closure is correct for final code generation. However, in the case of a first-order call, the interface is dependent on which input arguments are *needed*. As every closure procedure has a corresponding non-closure procedure, the transformation can replace the higher-order call to *proc* with a first-order call to $proc'$, where $proc'$ is the non-closure version of *proc*.

This optimisation will replace $proc{<}c_1, \ldots, c_n{>}(arg_1, \ldots, arg_m)$ with the first-order call $proc'(c_1, \ldots, c_n, arg_1, \ldots, arg_m)\langle ins, outs\rangle$, where $\langle ins, outs\rangle$ are the global flows of the $proc'$ procedure.

Reducing a call from a higher-order call to an equivalent first-order call is beneficial for several reasons. While a first-order call has fewer associated overheads, first-order calls are amenable to many more optimisations, including the full benefits of neededness (with the removal of un-needed arguments), and inlining. First-order calls are also annotated with information on exactly which global flows the procedure has, which is beneficial in the global variable optimisations in Section 7.2.

7.2 Optimisation of Global Variable Manipulation

In this section, we introduce analyses and transformations to optimise the use of global variables in a program represented in LPVM. These optimisations remove unnecessary

reading from and writing to of global variables as these may produce redundant overhead. We also further extend the neededness analysis as discussed in Section 7.1.1.

We define two optimisations to remove `load` and `store` instructions where possible in forwards (Section 7.2.1) and backwards (Section 7.2.2) passes, as an extension to the existing body builder. Further, we define an optimisation that constrains the global flow interface of procedures (Section 7.2.3), allowing for stronger optimisations in future body builder passes.

7.2.1 Forward Analysis and Transformation

In a forward analysis, the body of an LPVM procedure is traversed in the order of execution. Due to the property of forks in LPVM being pairwise disjoint, a forwards analysis analyses each fork individually. As such, analyses and transformations are applied up to the end of a body, diverging at a fork if present.

In this analysis, the body builder maintains a single map called the *global value map*, m. The global value map associates a global variable to some value. This map is maintained throughout this analysis and transformation to include currently known values for global variables. Through the analysis LPVM `load` and `store` instructions, *def-use* chains for the referenced global variable are inferred, with the map m marking the currently defined value for corresponding global variable.

Figure 7.1 presents the analysis and transformation that occurs for this optimisation. The rule F initialises the analysis with an empty map. F^m_{body} describes the analysis of a sequence of primitives terminated by a fork, passing the updated map, m', through the sequence of primitives. F^m_{prim} describes the transformation of a single primitive given the map m, and accordingly how the analysis modifies m.

To analyse an LPVM `load` instruction $\texttt{load}(\langle g \rangle, ?v)$, where some global variable, g, is loaded into a local variable, v, the analysis considers the map, m. If m does not contain a defined value for g, this instruction marks the start of a def-use chain. The transformation leaves the instruction unmodified and updates m to reflect the current known value of g to the now loaded value, v.

Otherwise, where m does contain an associated value, u, of the global variable, g, the `load` instruction is rendered redundant through some previous definition of the value

$$F[\![proc(\dots)<\dots> : body]\!] = proc(\dots)<\dots> : F^{\varnothing}_{body}[\![body]\!]$$

$$\begin{aligned}
F^m_{body}[\![\texttt{case}\ v\ \{b_1;\dots;b_n\}]\!] &= \left[\texttt{case}\ v\ \left\{F^m_{body}[\![b_1]\!];\dots;F^m_{body}[\![b_n]\!]\right\}\right] \\
F^m_{body}[\![[\,]]\!] &= [\,] \\
F^m_{body}[\![[prim \mid prims]]\!] &= prim' + F^{m'}_{body}[\![prims]\!] \\
&\quad \text{where } (m', prim') = F^m_{prim}[\![prim]\!]
\end{aligned}$$

$$\begin{aligned}
F^m_{prim}[\![\texttt{load}(\langle g\rangle, ?v)]\!] &= \begin{cases} (m[g \mapsto v], [\texttt{move}(u, ?v)]) & \text{if } m(g) = u \\ (m[g \mapsto v], [\texttt{load}(\langle g\rangle, ?v)]) & \text{otherwise} \end{cases} \\
F^m_{prim}[\![\texttt{store}(v, \langle g\rangle)]\!] &= \begin{cases} (m, [\,]) & \text{if } m(g) = v \\ (m[g \mapsto v], [\texttt{store}(v, \langle g\rangle)]) & \text{otherwise} \end{cases} \\
F^m_{prim}[\![\texttt{foreign}\ proc(\dots)]\!] &= (m, [\texttt{foreign}\ proc(\dots)]) \\
F^m_{prim}[\![proc(\dots)\langle ins, outs\rangle]\!] &= (m \setminus outs, [proc(\dots)\langle ins, outs\rangle]) \\
F^m_{prim}[\![f(\dots)]\!] &= \begin{cases} (\varnothing, [f(\dots)]) & \text{if } resourceful(f) \\ (m, [f(\dots)]) & \text{otherwise} \end{cases}
\end{aligned}$$

FIGURE 7.1: Forward analysis and transformations of global variable manipulation in LPVM.

of g. As m contains a known definition of g, the `load` instruction will be loading an equivalent value as u. The instruction is thus removed, being replaced by a $\texttt{move}(u, ?v)$ instruction, and does not define a new def-use chain, as exemplified in Listing 7.1. Together with value propagation, all further instances of v can be replaced by u, and this `move` instruction may be removed entirely if value propagation causes v to become un-needed.

```
1 foreign lpvm load(<g>:int, ?u:int)
2 # intervening primitives
3 foreign lpvm load(<g>:int, ?v:int)
```

(A) Before optimisation.

```
foreign lpvm load(<g>:int, ?u:int)
# intervening primitives
foreign lpvm move(u:int, ?v:int)
```

(B) After optimisation.

LISTING 7.1: Transformation of a `load` instruction into a `move` with forwards analysis, assuming the global variable, `g`, is untouched in the intervening primitives.

A `store` instruction, $\texttt{store}(v, \langle g\rangle)$, semantically, writes a value, v, to some global variable, g. In effect, the `store` instruction starts a new def-use chain for g, with the value of g being that of u until further redefined.

If m contains an associated value, u, for the global variable g, the analysis may find that the `store` instruction is redundant. If the previously defined value u is equivalent to the newly defined value v, the analysis finds this `store` instruction to be redundant.

This case can arise in two ways: either some previous `load` instruction has loaded the value u from g and the `store` instruction now writes u back into g (as in Listing 7.1), or two successive `store` instructions have attempted to write the same value to g. In either of these cases, the transformation can remove the `store` instruction, removing the def-use chain that spawns from this instruction.

Otherwise, the analysis cannot show that the current value of g is the same as the new value v. As such the *forward* optimisation cannot remove the `store` instruction, and we define a new def-use chain for g, updating the map to reflect this newly defined value, v. This `store` may be removed in a backwards pass (Section 7.2.2) if the value written is never read before being overwritten, however, we cannot prove that it is redundant in a forwards analysis.

```
1 foreign lpvm load(<g>:int, ?u:int)
2 # intervening primitives
3 foreign lpvm store(u:int, <g>:int)
```

(A) Before optimisation.

```
foreign lpvm load(<g>:int, ?u:int)
# intervening primitives
```

(B) After optimisation.

LISTING 7.2: Removal of a `store` instruction with forwards analysis, assuming the global variable, `g`, is untouched in the intervening primitives.

This analysis is unable to remove any further instructions. However, the information contained in the global flows of other instructions is essential to the analysis. For a procedure to have an outwards global flow, for some global variable, g, the primitive may have the effect of overwriting the current value of g. The analysis conservatively assumes that each primitive with such a global flow for g does overwrite the value contained in g, ending a known def-use chain for g. As such, for all global variables with such an outwards global flow, each global variable and associated value must be removed from m.

For the remaining `foreign` primitives, this does not remove any global variables from the map, as only the LPVM `load` and `store` instructions have any global flows. For first-order calls, the annotated global variables in the outward global flow set are removed, as their defined values *may* change.

In the case of higher-order calls ($f(\dots)$ in Figure 7.1), if the called value is resourceful, as annotated in the higher-order type, all global variables in the global variable map must be removed, and otherwise, the map remains unchanged. Higher-order types lack annotation of which resources, and hence global variables may be modified, the analysis must conservatively assume that all global variables may have been changed by the higher-order call. However, with the call lowering optimisation, the analysis may be able to know the global flows of such a higher-order call, if it can first be transformed into a first-order call.

In the case that a primitive does not contain an outward flow for some global variable, g, the primitive may internally overwrite the value of the global variable. However, due to the callee save convention used for global variables, if there is no outwards global flow for g, the primitive must ensure the value of the global variable is saved, remaining unchanged immediately before and after a call to the primitive. In such a case, if a value is currently known for g, then the value of g after the primitive must be the same, and any currently known definition of g remains valid.

7.2.2 Backwards Analysis and Transformation

Backwards analyses are performed in reverse execution order. As with forwards analyses, due to the property that forks are pairwise disjoint, each fork is analysed independently. However, unlike forwards analyses, where the analysis can diverge from a fork onwards, a backwards analysis combines the analysis from each fork before analysis of the parent body.

This analysis is concerned with removing `store` instructions where the value written by the instruction is never read. As only the effect of the final `store` instruction in a sequence of `store` instructions to the same global variable dictates the value of the global variable, only the final `store` instruction is required, assuming no intervening primitive requires the stored value.

Throughout the analysis, a single set per fork is maintained, s. This set s contains global variables that have not been read via some `load` instruction or otherwise since being defined by some `store` instruction or a primitive with an outwards global flow. The

analysis uses this set to know when successive instructions can be removed, assuming any outwards global flows are never used.

We omit the details of neededness in this analysis as covered in Section 7.1.1. The analysis presented here assumes that all procedures are, unless otherwise stated, unneeded for simplicity. Within the compiler, this analysis is performed in tandem with neededness analysis, and we cannot omit procedures that are either non-pure or have some needed output.

The extended concept of neededness in the context of global variables is made concrete with the use of the set s. If all global variables with outwards flows are contained within the s, we consider the primitive unneeded. Effectively, any global variables that may be overwritten by this primitive are overwritten by some later primitive, as indicated by their membership in s, so the effects of the primitive are nullified.

The analysis and transformations are shown in Figure 7.2. The B rule describes the analysis and transformation of a procedure's body and initialises the map m to the empty set. B^s_{body} describes the analysis and transformation of a sequence of primitives, traversing the sequence in reverse and threading through the updated set, s', backwards through the sequence. B^s_{prim} describes the transformation of a single primitive, modifying the set s accordingly.

For an LPVM `store` instruction, `store`$(v, \langle g \rangle)$, semantically, some value, v, is written to the global variable, g. Where this global variable, g, is a member of the set s, the global variable has been overwritten by a successive `store` instruction, but has not been loaded. In this case, this earlier store instruction is redundant, and can be removed, as seen in Listing 7.3. We do not modify s in this case.

Otherwise, where s does not contain g as a member, the instruction writes a value to g which is then not overwritten by a further `store` instruction without the value being loaded in between. The transformation does not remove this `store` instruction and adds g to s.

All other `foreign` instructions remain unchanged. However, the set s can be modified. The analysis removes all global variables from the set where the global variable flows into the primitive. For LPVM `load` instructions, the loaded global variable will be removed, as this marks a usage of any earlier stored value. For all other `foreign` instructions,

$$B[\![proc(\ldots)<\ldots> : body]\!] = proc(\ldots)<\ldots> : body' \quad \text{where } (s, body') = B^{\varnothing}_{body}[\![body]\!]$$

$$B^s_{body}[\![[\texttt{case}\ v\ \{b_1;\ldots;b_n\}]]\!] = (\bigcap_{i\in 1\ldots n} s_i, [\texttt{case}\ v\ \{b'_1;\ldots;b'_n\}]) \quad \text{where } (s_i, b'_i) = B^s_{body}[\![b_i]\!]$$

$$B^s_{body}[\![[\,]]\!] = (s, [\,])$$

$$B^s_{body}[\![[prim \mid prims]]\!] = (s'', prim' + prims') \quad \text{where } (s', prims') = B^s_{body}[\![prims]\!],\ (s'', prim') = B^{s'}_{prim}[\![prim]\!]$$

$$B^s_{prim}[\![\texttt{store}(v, \langle g\rangle)]\!] = \begin{cases} (s, [\,]) & \text{if } g \in s \\ (s \cup \{g\}, [\texttt{store}(v, \langle g\rangle)]) & \text{otherwise} \end{cases}$$

$$B^s_{prim}[\![\texttt{load}(\langle g\rangle, ?v)]\!] = (s \setminus \{g\}, [\texttt{load}(\langle g\rangle, ?v)])$$

$$B^s_{prim}[\![\texttt{foreign}\ proc(\ldots)]\!] = (s, [\texttt{foreign}\ proc(\ldots)])$$

$$B^s_{prim}[\![proc(\ldots)\langle i, o\rangle]\!] = \begin{cases} (s, [\,]) & \text{if } o \subseteq s \\ ((s \cup o) \setminus i, [proc(\ldots)\langle i, o\rangle]) & \text{otherwise} \end{cases}$$

$$B^s_{prim}[\![f(\ldots)]\!] = \begin{cases} (\varnothing, [f(\ldots)]) & \text{if } resourceful(f) \\ (s, [f(\ldots)]) & \text{otherwise} \end{cases}$$

FIGURE 7.2: Backwards analysis and transformations of global variable manipulation in LPVM.

```
1 foreign lpvm store(u:int, <g>:int)
2 # intervening primitives
3 foreign lpvm store(v:int, <g>:int)
```

(A) Before optimisation.

```
# intervening primitives
foreign lpvm store(v:int, <g>:int)
```

(B) After optimisation.

LISTING 7.3: Removal of a `store` instruction with backwards analysis, assuming the global variable, `g`, is untouched in the intervening primitives.

the analysis does not remove any global variables from the set, as all other `foreign` instructions have no global flows.

First-order calls have an explicit annotation that denotes exactly which inwards (i) and outwards (o) global flows a procedure may have. If the analysis has found that $o \subseteq s$, we can remove the instruction, leaving s unmodified. This handles the case where the procedure has no *needed* outwards global flows, as all global variables overwritten by the primitive are overwritten by a later instruction, as marked by their membership in s.

Otherwise, as we cannot remove the primitive, the analysis uses these global flows to update the set s to include global variables that have an outward flow and no global variable with an inward flow, *i.e.*, $(s \cup o) \setminus i$. This removes all global variables from the set where the value *may* be read, and as such earlier instructions that overwrite the global variable are required. The analysis also includes all global variables in o that are not in i. These global variables are overwritten by the primitive, and as such, unless it is otherwise read, earlier `store` instructions are redundant.

For higher-order calls ($f(\dots)$), the higher-order call is left unmodified. As the higher-order term called does not annotate exactly which global variables the term may modify, only if the term may modify some global variable, be it with an inwards or outwards flow. In such a case, the analysis conservatively assumes that all global variables may have flowed in, being read in the call, and removes all global variables from s, resulting in $\varnothing$. As in Section 7.2.1, if this call can be lowered to a first-order call, the analysis can be provided with more information regarding exactly which global variables flow into the call.

7.2.3 Global Flow Interface Analysis and Transformation

Following the forwards and backwards analyses and transformations previously defined, it is possible that the global flow interface of a procedure can be reduced. *I.e.*, there may be global flows that do not occur in the procedure, yet occur in the interface. This is possible through the removal of a `load` or `store` instruction never occurring within the body of a procedure, nor in effect through calling some other procedure.

Reducing the global flows is beneficial, as reduced global flow sets allow for stronger optimisations within other analyses. This includes the removal of other `load` and `store` instructions and the removal of first-order calls where it is known there is no outwards global flow.

In this analysis and transformation, we define the intersection ($\cap$) of two global flows as the intersection of their component sets of global variables, respectively. That is, $\langle in_1, out_1\rangle \cap \langle in_2, out_2\rangle \coloneqq \langle in_1 \cap in_2, out_1 \cap out_2\rangle$. Similarly, we define the union ($\cup$) of two global flows as the union of their component sets of global variables.

The goal of the analysis is to collect all *effective* global flows throughout the procedure. Effective global flows are those global flows that can be seen through a call to a procedure, *i.e.*, if a global variable is read or overwritten. As the interface of a procedure already dictates effective global flows of a procedure, the current interface of a procedure is taken into account for the analysis and corresponding transformation. The analysis and transformation in place in this optimisation can be found in Figure 7.3.

$$I[\![proc(...)<in, out> : body]\!] = proc(...)(F \cap <in, out>) : body$$
$$\text{where } F = I^{in \cap out}_{body}[\![body]\!]$$

$$I^G_{body}[\![[\texttt{case } v \,\{b_1; \ldots; b_n\}]]\!] = \bigcup_{i \in 1 \ldots n} \langle in_i, out_i \rangle \cup \langle G \cap G', \varnothing \rangle$$
$$\text{where } \langle in_i, out_i \rangle = I^G_{body}[\![b_i]\!]$$
$$G' = \bigcup_{i \in 1 \ldots n} out_i \setminus \bigcap_{i \in 1 \ldots n} out_i$$
$$I^G_{body}[\![[\,]]\!] = \langle \varnothing, \varnothing \rangle$$
$$I^G_{body}[\![[prim \mid prims]]\!] = \langle in, out \rangle \cup \langle in' \setminus out, out' \rangle$$
$$\text{where } \langle in, out \rangle = I_{prim}[\![prim]\!]$$
$$\langle in', out' \rangle = I^G_{body}[\![prims]\!]$$

$$I_{prim}[\![proc(\ldots)\langle in, out \rangle]\!] = \langle in, out \rangle$$
$$I_{prim}[\![\texttt{load}(\langle g \rangle, ?v)]\!] = \langle \{g\}, \varnothing \rangle$$
$$I_{prim}[\![\texttt{store}(v, \langle g \rangle)]\!] = \langle \varnothing, \{g\} \rangle$$
$$I_{prim}[\![\texttt{foreign } proc(\ldots)]\!] = \langle \varnothing, \varnothing \rangle$$
$$I_{prim}[\![f(\ldots)]\!] = \begin{cases} \langle \mathbb{U}, \mathbb{U} \rangle & \text{if } resourceful(f) \\ \langle \varnothing, \varnothing \rangle & \text{otherwise} \end{cases}$$

FIGURE 7.3: Analysis and transformations of global flow interfaces in LPVM.

The analysis begins by traversing the body of a procedure with global flows $\langle ins, outs \rangle$, backwards with a set, $G = in \cap out$. The set G represents global variables that have both an inwards and outwards global flow in the current interface and is used throughout the analysis of the body, specifically in the analysis of flows across branches of forks.

In traversing backwards, the global flows across all branches of a fork must be combined such that the resultant set represents the global flows that occur across all branches. Naïvely, one may decide to take the union across the flows analysed across all branches. However, this may incorrectly restrict the behaviour of some fork's global flows wherein the branches have differing global flows.

```
1 foo(b, x)<{g}, {g}>:
2     case b:
3     0:
4         # nothing
5     1:
6         foreign lpvm store(x, <g>)
```

LISTING 7.4: Branches in LPVM with differing global flows.

Consider the example in Listing 7.4. In the `1` branch, there is an outwards global flow of the variable `g`, however there is no global flow of `g` in the `0` branch. Taking the union of these global flows results in the global flows $\langle \varnothing, \{\mathtt{g}\} \rangle$.

According to the current interface, the value of `g` prior to the call may be required and the value may be overwritten. While the value prior to the call is never read, it is never overwritten in the `0` branch, which follows the semantics of the current global flows, $\langle \{\mathtt{g}\}, \{\mathtt{g}\} \rangle$, and not $\langle \varnothing, \{\mathtt{g}\} \rangle$ and `g` is not overwritten. The `1` branch, however, is semantically correct under the $\langle \varnothing, \{\mathtt{g}\} \rangle$ global flows. The only global flows that are compatible with both branches is $\langle \{\mathtt{g}\}, \{\mathtt{g}\} \rangle$, even though the value of `g` is never loaded.

Due to the nature of the definition of global flows, such cases can only arise wherein the procedure has an inwards and outwards flow of some global variable, else the procedure would have an outwards flow for such a global variable within the body. Such flows can be analysed by finding some outwards global flow that occurs in *some* but not *all* branches across a fork.

This is the purpose of the set G' in the case of I_{body}. Along with the union of all global flows that occur across all branches, $\langle in_i, out_i \rangle$, the analysis includes an inwards global flow for all global variables that occur in G and flow outwards in some branch, but not all branches. While this set is a weaker constraint of the global flows across the fork, these global flows are correct under the original interface.

Conversely, if all branches have the same, outwards only, global flow for some global variable, the global flows that do occur across the branch do not require the inclusion of inwards flows of such global variables. An example of this case can be seen in Listing 7.5 with respect to the global variable `g`. As all branches overwrite the value of `g`, the analysis can conclude that the outwards global flow does occur with no inwards flow.

```
1 bar(b, x, y)<{g}, {g}>:
2     case b:
3     0:
4         foreign lpvm store(x, <g>)
5     1:
6         foreign lpvm store(y, <g>)
```

Listing 7.5: Branches in LPVM with identical global flows.

For the flows between a primitive, *prim*, and the remaining primitives in the rest of a body, including some optional fork, *prims*, the analysis includes all global flows that occur within the primitive, $I_{prim}[\![prim]\!]$. Further it excludes all flows that are *killed* in the flows of *prims*, $\langle in', out' \rangle$, by *prim i.e.* all global variables that flow out of *prim*.

A global variable that flows out of some primitive, but not in, has the effect of all further inwards flows of the same global variable being irrespective of the value of the global variable before the primitive. *I.e.*, if the global variable is read by some primitive in *prims*, the state of the global variable does not depend on an inwards global flow. In such cases, we can remove such inwards global flows from the resultant global flows.

Once the body has been traversed backwards, the analysis results in the global flows F. Here, the optimisation transforms the global flows from the procedure from $\langle in, out \rangle$ into $F \cap \langle in, out \rangle$. This removes all global flows that do not occur within the procedure and are known to be transparent to the caller from the definition of the global flows sets at the creation of this procedure.

For example, in Listing 7.5, the analysis will show that both branches exhibit the same global flows, $\langle \varnothing, \{\texttt{g}\} \rangle$, with the fork and entire procedure body having the same global flows throughout. The analysis has shown that the value of `g` prior to a call to `bar` is overwritten, and as such the interface of the procedure is transformed into $\langle \varnothing, \{\texttt{g}\} \rangle$.

In Listing 7.4, due to the in and out flow of `g` in the current interface of `foo`, the global flows across all branches is restricted to $\langle \{\texttt{g}\}, \{\texttt{g}\} \rangle$. This results in the entire body requiring the same such global flows, and the interface of the procedure remains unchanged after the transformation.

The global flows inferred through this analysis do not necessarily represent the most constricted global flows that could occur across all procedures. Much like with neededness analysis (Section 7.1.1), finding the optimal (*i.e.*, the smallest) global flows would require finding a fixed point across all procedures. This fixed point would be reached where the global flows cannot be reduced further across any procedure without violating the correctness of global flows.

Such a fixed point can be found, however, the computation of such a fixed point is computationally expensive by its very nature. Empirically, across the test suite used to ensure the correctness of the compiler, finding such a fixed point does not further reduce the global flows across the tests. As such, we have chosen to omit the full fixed point of this analysis from the implementation within the Wybe compiler, much like with neededness.

Chapter 8

Translation to LLVM

This chapter details the translation of LPVM into *Low-Level Virtual Machine* [4]. We detail the translation of a Wybe module, translating the introduced elements of LPVM into Low-Level Virtual Machine (LLVM).

LLVM is the backend intermediate language used by the Wybe compiler. LLVM has been used as the backend for many widely used compilers, such as the Haskell compiler, GHC [57], and the C language family's compiler, Clang. We also leverage LLVM here due to its attractive properties. These properties include the translation into machine code for many target architectures and a many available optimisation passes.

Broadly speaking, the translation from LPVM into LLVM translates arguments into operands, procedures into functions, statements into instructions, and forks into branch instructions. LLVM is an SSA form intermediate representation, and as such requires versions of variables to be merged via a ϕ-node at the beginning of each basic block. Due to all branching in LPVM representing diverging branches in the computation, no ϕ-nodes are required to be introduced in all translated LPVM procedures, greatly simplifying the translation. The LLVM compiler may choose to insert ϕ-nodes throughout its internal optimisations, however.

8.1 LLVM Types

Like most intermediate representations, LLVM does not have a native mechanism for multiple return values. In the existing translation into LLVM, the compiler translates

the returned values into a single return value, with the return type being dependent on the number of return values. For procedures with no out-flowing return value, the return type of the translated function is `void`. Procedures that have a single return value also have a single return value of the corresponding LLVM type.

Where a procedure has multiple output values, the corresponding LLVM function can only have one return value. This value is an aggregate type, called a *tuple type* that contains sequential fields for each return type. The return value of the function is constructed by sequentially inserting each operand for the associated out-flowing value.

LLVM also supports higher-order types called *function types*. Like functions in LLVM, function types have two components, a return type and a list of parameter types. In Wybe and LPVM, the equivalent type is a procedure type with a collection of modifiers and a list of types and corresponding flows.

The compiler translates a procedure type into a corresponding function type where the input parameters correspond to the in-flowing procedure arguments, and the return type is translated with the outwards flowing arguments as either a `void` type, a single type, or a structure type for more than one return type.

8.2 Closures

In Chapter 6 during unbranching, the compiler translated each procedure reference into a reference to a closure procedure. These closure procedures have a key property: excluding any parameters that pass free variables, all parameters could not be marked as *unneeded*. In all other procedures, if a parameter is marked as unneeded, it does not get translated into a parameter in the corresponding LLVM function definition.

As unneeded parameters do not get translated into parameters in the corresponding LLVM function, if non-free closure parameters were marked as uneeded, the interface that defined how arguments are passed between caller and callee would be violated. As the callee cannot know if an argument is needed or not, the callee may pass additional arguments to the caller, causing the callee to receive the arguments in a different order

than expected, violating the argument passing interface. Marking all non-free parameters as needed ensures that such a case cannot occur, and if the parameter were not needed, it can simply be ignored.

Translating a closure procedure requires additional preprocessing before the regular procedure translation process can occur. The compiler removes all *free* parameters and replaces them with a single parameter that will be used to pass the closure's *environment*, `env`. We elect to use a flat environment representation, representing the environment as a vector of free variables, due to the relative efficiency when accessing the free variables and ease of construction.

```
1 anon(^b:bool, p1:int, p2:int, ?p3:int)<{}, {}>:
2     case b:bool:
3     0:
4         foreign lpvm move(p1:int, ?p3:int)
5     1:
6         foreign lpvm move(p2:int, ?p3:int)
7
8 clos(^b:bool, p1:int, p2:int, ?p3:int)<{}, {}>:
9     anon(b:bool, p1:int, p2:int, ?p3:int)
```

(A) Before closure preprocessing.

```
1 anon(^b:bool, p1:int, p2:int, ?p3:int)<{}, {}>:
2     case b:bool:
3     0:
4         foreign lpvm move(p1:int, ?p3:int)
5     1:
6         foreign lpvm move(p2:int, ?p3:int)
7
8 clos(env:int, p1:int, p2:int, ?p3:int)<{}, {}>:
9     foreign lpvm access(env:int, 8:int, 8:int, 0:int, ?b:bool)
10    anon(b:bool, p1:int, p2:int, ?p3:int)
```

(B) After closure preprocessing.

LISTING 8.1: Example closure procedure before and after preprocessing for translation to LLVM, continuing from Listing 6.4, with changes highlighted in red. The parameters preceded by a ^ are closed variables.

The body of a closure procedure is also modified. For each needed free parameter, the compiler prepends an LLVM `access` instruction to the body, which is used to access some data stored at some memory address at a given offset. Each needed free parameter,

p_i, corresponds to an `access` instruction that loads index $i*8$ (accounting for word size) from `env`. An example of such preprocessing can be found in Listing 8.1.

In parallel to the preprocessing step of a closure procedure, the compiler translates procedure references. A procedure reference is a reference to an LPVM procedure along with a list of closed arguments. These closed arguments will correspond directly to a free parameter in the closure procedure.

In translating a procedure reference to procedure p with associated closed arguments, $arg_1, \ldots, arg_n$, the compiler constructs an LLVM array of size $n+1$. This array is filled with a pointer to the procedure along with the values of the arguments.

If all arguments are constant values, the compiler allocates a global constant for the array and uses a pointer to this global constant as the LLVM operand. Otherwise, the closure is allocated using heap memory large enough to store the array and uses a pointer to this memory as the LLVM operand.

Both the dynamic and static allocation of closures and their environments in this way solve the downwards and upwards funarg problems [12, 13]. Here the compiler is able to reuse some LLVM operands in the case some closure is identical to a previously generated closure, be it heap-allocated or otherwise [19].

8.3 Higher-Order Calls

All valid higher-order calls in LPVM call a value that is either a variable with some higher-order type or a procedure reference with accompanying closed values. The latter case can be excluded in the translation to LLVM due to the optimisation pass outlined in Section 7.1, transforming all higher-order calls to procedure references into first-order calls. Regardless, the translation of such a higher-order call to some procedure reference would simply translate a first-order call to the referenced procedure.

All other higher-order calls will be to some variable. This variable will be a reference to some closure structure that was created via the translation process as outlined in Section 8.2. A higher-order call in LLVM requires an operand of a function pointer type along with the arguments to the call. The closure variable contains this function pointer. All closures contain the function pointer as the first value of the array, and

as such can be accessed uniformly via the zeroth index of the array. The higher-order call then, additionally, passes the closure variable as the first argument to the call. This passes the environment to the closure, which then allows access to the free variables for the referenced procedure. To support this, the compiler augments all higher-order procedure types with an additional word-sized integer argument as the first parameter type (`i64` in LLVM on 64-bit machines).

A calling convention dictates how arguments are passed from caller to callee and vice versa. The calling convention decides which arguments are passed via registers and which are passed via the stack. In the case of first-order calls, the compiler uses the `fastcc` calling convention. This calling convention aggressively attempts to pass all arguments and return values in registers as opposed to on the stack. Due to some architectures passing different variables in different registers and the effects of different calling conventions, the compiler modifies the arguments of higher-order calls to ensure that arguments are passed in the same arguments regardless of the type.

Before making a call, the compiler passes all arguments cast to a word size integer (*i.e.*, `i64` in LLVM) through zero extensions (padding with zero bits) and *bitcasts* to ensure that the width of the type matches that of a word-sized integer. A bitcast circumvents LLVM's strict typing, allowing the value to be used as a word-sized value. This same process is performed currently to support generic types, with all arguments that are generic cast to the respective word-sized integer type.

8.4 Global Variables

LLVM, like most imperative intermediate representations, supports global variables. Global variables are assigned a unique name and can be used by referring to this name as you would a regular pointer value.

Global variables in LLVM have a uniform name and type throughout all instances of their usage in any given LLVM module, as is documented in their declarations. For each global variable, the compiler uses the underlying resource's fully module-qualified name and corresponding LLVM type for the global variable.

LLVM supports various *linkages* for global variables that define how the linker handles the resolution of a global variable's address. The compiler requires the `external` linkage. This linkage allows for a global variable to be visible in any other module that may use it. Global variables with this linkage are used to resolve external references to this variable.

LLVM optionally allows a global variable to have an initial value. Where a global variable is referenced in some module, but not defined, the compiler does not provide an initial value. Otherwise, if the global variable is defined in the given module, the compiler provides an initial value of `undef`, which represents some unspecified value. As the initialisation of some resource, and hence some global variable, occurs within the module where it is defined, the compiler does not require a specific value to initialise the global variable.

In LLVM, global variables are referenced by name as a pointer to the underlying global variable. As with any other pointer type in LLVM, the LLVM `load` and `store` instructions are used to load the value of and store some value to the global variable respectively. The LLVM `load` and `store` instructions parallel these directly, providing a translation of these instructions as shown in Listing 8.2.

```
1 foreign lpvm load(<global>:type, ?var:type)
2 foreign lpvm store(val:type, <global>:type)
```

(A) LPVM instructions.

```
%var = load type, type* @global
store type %val, type* @global
```

(B) LLVM instructions.

LISTING 8.2: Equivalent LPVM and LLVM instructions to manipulate global variables.

As *phantom* types have a zero-bit representation, there is no equivalent type in LLVM that is valid for a variable. As such all global variables that have a phantom type are excluded, and any `load` and `store` instructions that manipulate such global variables are not translated into LLVM instructions. This omission is safe to perform, as *phantom* typed values in Wybe and LPVM exist primarily to enforce the ordering of calls declaratively, because *phantom* types cannot be used to transmit information. As LLVM does not re-order or omit calls that it cannot prove can safely be omitted or re-ordered in the suite of optimisations the LLVM compiler performs, the ordering of calls is still ensured.

Chapter 9

Evaluation of the Extended Wybe Language

In this chapter, we outline the evaluation of the extended higher-order Wybe language in comparison to the previous, first-order Wybe language. We outline the programs that are used throughout the experiments, and the design and results of these experiments. Further, we analyse the results we have found.

This evaluation investigates the effects of both the use of higher-order programming in comparison to the use of first-order programming and separately the globalised resource implementation strategy in comparison to the parameterised resource implementation strategy.

9.1 Evaluation Environment

Throughout this evaluation, we make use of two versions of the Wybe compiler. The first is a version of the compiler that existed before the extensions we have made to the language, the first-order Wybe compiler. The second is a version of the compiler with all extensions implemented for the Wybe language, the higher-order Wybe compiler. [1]

[1] The first-order and higher-order versions of the Wybe compiler used in this evaluation can be found at https://github.com/pschachte/wybe/tree/d2b5fbc500 and https://github.com/pschachte/wybe/tree/1655a61d68, respectively.

The version of the LLVM compiler the Wybe compiler used was LLVM version 10.0.0 targeting the `x86_64-pc-linux-gnu` architecture.

The evaluation was performed on a Dell XPS™ 13 9360, with an Intel® Core™ i7-8550U CPU @ 1.80GHz–2.00GHz and 8GB RAM. The programs were executed within the Windows Subsystem for Linux 2, using an Ubuntu™ 20.04.3 LTS distribution.

9.2 Benchmark Wybe Programs

Throughout these experiments, we will compare the runtime and executable size of pairs of programs as listed in Table 9.1. Full listings of these programs are provided in Appendix B. Each program was written first with the use of first-order code, and later generalised with the use of higher-order code. The generalisation makes use of general-purpose higher-order procedures, such as `map`, with distinct logic extracted and parameterised. Some of these programs are also present in similar language benchmarking studies, such as the *Mandlebrot* program [8, 58, 59] and the *N Body* program [8, 58, 60], being present in a common programming language suite, the Computer Language Benchmarking Game [61].

Program	Description	Listings
Fibs	Sum of Fibonacci numbers	Appendix B.1
Knapsack	Solution to the knapsack problem	Appendix B.2
Mandlebrot	Generation of the Mandlebrot set	Appendix B.3
N Body	Simulation of the N Body problem	Appendix B.4
Sonar Sweep	List traversal problem	Appendix B.5
Sort	Sorting various data types	Appendix B.6

TABLE 9.1: Suite of benchmark programs.

These programs have been chosen to reflect different use cases of higher-order programming and resource usage. The *Fibs* program performs a relatively small amount of higher-order calls, and instead, the term called performs the bulk of the computation. In contrast to this, the *Mandlebrot* program performs a relatively high number of higher-order calls, yet the higher-order term itself performs a relatively small amount of computation. Some programs, such as the *Mandlebrot*, *N Body* and *Sort* programs only make use of the `io` resource, whereas the *Fibs*, *Sonar Sweep*, and *Knapsack* programs make use of resources to compute a mathematical or optimisation result.

The inputs for each implementation of a program are identical. The inputs were chosen, being randomly generated where possible, to ensure that the run times across programs were comparable. The inputs chosen were designed to have a runtime around the same magnitude across all programs, and above 1 second across all tests. This threshold further ensures the stability of runtimes across tests. The exact inputs are omitted from this work due to their large size and unimportance to the work itself.

9.3 Program Size

Higher-order code has the potential to increase the expressiveness available to the programmer. This, in general, can allow for programs to make greater reuse of procedures that can be generalised to take procedure arguments, modifying the internal behaviour of a procedure depending on what procedure argument was passed.

The use of global variables also has ramifications for the resultant compiled code. While the promotion of resources to global variables is transparent to an end-user, the compiled code is different, requiring alternative instructions to manipulate global variables.

In these experiments, we are interested in investigating the separate effects on the size of programs. We choose to investigate the effects on program size with three metrics: object file size, executable size, and source lines of code (SLOC). Due to the compiler being deterministic, these metrics are static, requiring a single compilation for each program implementation. We will perform two experiments to control for the effects of either the usage of higher-order programming or globalised resources.

The object file size is generated by the compiler for incremental compilation, being reused if no source changes were made. This file contains information that is relevant to a single Wybe module, containing both the LPVM and LLVM representations of the module. The executable file conversely contains all code that is required to execute the program, containing code from not only the concerned module, but various libraries, with debugging and unused symbols stripped in the creation of the file.

We measure the SLOC of a program by counting the number of lines of code in the source file, sans comments, and blank lines. To ensure that the formatting does not affect the

SLOC of a program, we require that the formatting of code is consistent between the pair of implementations of each program.

9.3.1 Program Size and Higher-Order Programming

In investigating the effects of higher-order programming, both program implementations will be compiled with the extended Wybe compiler. This has the effect of controlling for the usage of global variables, as both programs will make use of globalised resources.

The baseline in this experiment will be the first-order implementation. We hypothesise that there will be mixed effects on the program size. The increased expressiveness in these programs allows for greater code reuse in some programs, and we expect to see reductions in metrics there. In programs without greater code reuse, we expect to see a small increase due to higher-order programming introducing additional procedures and executable code.

9.3.2 Program Size and Resource Implementation Strategy

As higher-order programming is not possible with the first-order Wybe compiler, to investigate the effects of globalised resources, we will use the first-order program implementations, with the programs being compiled with both the first-order and higher-order Wybe compilers. This also has the effect of controlling for the effects of higher-order programming, as both compiled programs will be identical. As the programs are identical, there is no information gained in investigating the SLOC.

This experiment's baseline is the programs compiled with the first-order compiler. This compiler makes use of parameterised resources. Ideally, the use of either resource implementation should not have great effects on the compiled code. We hypothesise that there will be small effects on the code size, with a general increase for globalised resources due to the additional instructions required for such resources.

9.3.3 Results and Analysis

9.3.3.1 Higher-Order Programming

In Table 9.2, the object size, executable size, and SLOC of each program are provided, for both higher-order and first-order implementations.

Program	Order	SLOC		Object Size (B)		Executable Size (B)	
Fibs	First	19		75 128		19 088	
	Higher	19	+0.0%	80 048	+6.5%	19 160	+0.4%
Knapsack	First	52		268 680		19 496	
	Higher	52	+0.0%	272 904	+1.6%	19 568	+0.4%
Mandlebrot	First	32		223 040		24 384	
	Higher	32	+0.0%	241 440	+8.2%	24 376	+0.0%
N Body	First	71		385 536		24 608	
	Higher	64	−9.9%	434 064	+12.6%	24 768	+0.7%
Sonar Sweep	First	31		107 496		19 400	
	Higher	31	+0.0%	113 256	+5.4%	19 472	+0.4%
Sort	First	134		463 736		29 800	
	Higher	37	−72.4%	179 360	−61.3%	29 488	−1.0%

TABLE 9.2: The size of object files, compiled executables, and number of source lines of code (SLOC) for first-order and higher-order implementations of programs. The approximate change in each metric is provided.

The SLOC across the *Fibs*, *Knapsack*, *Mandlebrot*, and *Sonar Sweep* programs do not change with respect to the usage of first-order and higher-order programming. This is due to the higher-order procedures in the higher-order implementations being used once, having an identical number of SLOC to their first-order counterparts.

The SLOC in higher-order programs can be reduced, however, if commonplace higher-order procedures are placed in a common module, say in the standard library. Examples of this are the `map` procedure defined for the generic `list` type in both the *Knapsack* and *N Body* programs.

Contrasting the four previous programs, the higher-order *N Body* and *Sort* implementations did see a reduction in the SLOC when compared to the first-order implementations.

The higher-order *N Body* implementation has a reduced SLOC due to the reuse of the `map` predicate, used twice in `step`. The first-order implementation requires two separate procedures, `map_update_v` and `map_update_pos` to perform the same actions. These

two procedures are very similar in structure, differing only in the procedure that is applied to each `body`.

The *Sort* program shows the greatest reduction in the SLOC of all programs, reducing nearly 3/4 of the SLOC. The *Sort* program implements a sorting algorithm over numerous types, with the comparison test used for each type is the only difference required for each type. In the first-order implementation, we are unable to factor this component out of the algorithm and are forced to duplicate the sorting algorithm per type required. If we also desire to change the sorting behaviour, say by sorting in reverse, we cannot simply change the comparison test but must duplicate the procedure also.

The higher-order implementation overcomes this limitation by parameterising the comparison procedure used in the sorting algorithm. As the program can now reuse the same sorting procedure per type, the SLOC is constant regardless of the number of types that require sorting. We can also change the comparison predicate without duplication of the sorting predicate.

The I/O performed in this program also has similar properties, with both the `reader` and `printer` procedures parameterised for a generic type, providing similar benefits to the sorting procedure.

Excluding the *Sort* program, across all programs the size of object files increases slightly. The change in object file size is caused by two effects, a change in the size of LPVM and LLVM sections in the files. Closures within a program create additional, albeit small, procedures, that lead to these increases in object file size.

In the case of the *Sort* program, the object file size decreases. This corresponds with the decrease in SLOC seen for the *Sort* program, as the decrease in the number of procedures leads to an overall decrease in the object file size. As with the SLOC, placing common higher-order procedures, such as the `map` procedure, into a common module in the standard library would further reduce the object file sizes, amortising the overall cost associated with an increase in object file size.

The executable size is quite small for all programs, which is expected due to these files containing only the necessary instructions to execute the program in a machine code. Excluding the *Sort* and *Mandlebrot* programs, there is a very small increase in the executable size, which is likely a result of the additional code required for higher-order

procedures, though the effect is very small. In the case of the *Sort* program, the decrease is likely a result of the same effects on SLOC and object file size, with greater re-use of common procedures.

9.3.3.2 Resource Implementation

The effects on program size from the resource implementation used are presented in Table 9.3. Note that the measure of SLOC is not documented here, as the programs are identical, regardless of the use of parameterised or globalised resources.

Program	Resource	Object Size (B)		Executable Size (B)	
Fibs	Parameters	69 808		18 720	
	Globals	75 128	+7.6%	19 088	+2.0%
Knapsack	Parameters	257 584		19 128	
	Globals	268 680	+4.3%	19 496	+1.9%
Mandlebrot	Parameters	233 576		24 064	
	Globals	223 040	−4.5%	24 384	+1.3%
N Body	Parameters	395 848		24 288	
	Globals	385 536	−2.6%	24 608	+1.3%
Sonar Sweep	Parameters	100 720		19 024	
	Globals	107 496	+6.7%	19 400	+2.0%
Sort	Parameters	464 832		29 480	
	Globals	463 736	−0.2%	29 800	+1.1%

TABLE 9.3: The size of object files, compiled executables for first-order program implementations with parameterised and globalised resources. The approximate change in each metric is provided.

Regarding the size of object files, we see mixed effects depending on the program. On average across all programs, we do see an overall increase in the size of object files. This likely corresponds to the additional instructions required for the manipulation of global variables.

The decrease in object file size for the *Mandlebrot*, *N Body*, and *Sort* programs with the use of globalised resources however is interesting. These programs primarily make use of the `io` resource, with the core of these programs not making use of resources at all. As the `io` resource has a *phantom*, type this global variable manipulation is not translated into LLVM, which reduces the number of parameters being passed and the number of instructions overall.

Across all programs, we see an increase in the executable size. We see the smallest increases in the executable sizes for the same programs that have reduced object file sizes, namely for the *Mandlebrot*, *N Body*, and *Sort* programs. This smaller increase is a result of the decrease in the number of instructions that manipulate the global variables present in these programs, due to instructions to manipulate *phantom-typed* resources not being present.

However, we do see a negligible increase overall, which is the result of executables requiring instructions to manipulate global variables. These instructions increase the executable size. Further, all programs when compiled require the use of built-in resources to handle the command-line arguments, which are translated into both global variables and some instructions for their initialisation.

9.4 Execution Runtime

The runtime of a program provides insight into the relevant efficiency of the program. The runtime of a program is also affected by the ability of the compiler to produce optimised code.

Higher-order programming has often been viewed as being less efficient than first-order code due to various overheads [8, 15, 62]. Higher-order programming in Wybe has such overheads, as higher-order calls require pointer dereferences to both make the higher-order call and retrieve closed variables from the closure's environment and memory allocations for closures.

The use of global variables may also have ramifications on the runtime of a program. Excessive reading and writing from memory can be slow, and as such redundant manipulation of global variables should be avoided in performant code.

9.4.1 Execution Runtime and Higher-Order Programming

To investigate the effects of higher-order programming on the runtime of Wybe programs, we perform a series of timed executions of each of the first-order and higher-order program implementations. Each program will be executed over a series of trials, to control for slight variations in runtime between trials of the same implementation.

To control for the effects of the presence of global variables, both programs will be compiled using the higher-order Wybe compiler.

9.4.2 Execution Runtime and Resource Implementation Strategy

In investigating the effects of global variables, a similar execution scheme will be used here, with both programs executed over a series of trials to control for variations in the runtime.

However, to control for effects on the runtime of higher-order programming, the programs will both be the first-order implementation, and instead, the programs will be compiled with the first-order and higher-order Wybe compilers. This controls for any effects of higher-order programming on the runtime.

9.4.3 Results and Analysis

9.4.3.1 Higher-Order Programming

In Table 9.4, statistics regarding the runtimes across first and higher-order program implementations are presented. These statistics are produced over a series of 30 trials, with a full table of runtimes found in Table A.1. Table 9.4 further shows the number of higher-order calls performed in the higher-order implementation of each program.

Program	First-Order (s)			Higher-Order (s)			Higher-Order Calls
	m	μ	σ	m	μ	σ	
Fibs	6.048	6.050	0.016	6.053	6.055	0.018	903
Knapsack	5.326	5.330	0.050	5.363	5.361	0.054	500 200
Mandlebrot	4.785	4.829	0.101	4.835	4.873	0.098	173 912 082
N Body	4.478	4.498	0.063	4.910	4.910	0.118	105 000 005
Sonar Sweep	7.473	7.487	0.045	7.427	7.427	0.004	39 760 695
Sort	3.450	3.471	0.084	3.568	3.578	0.053	25 413 807

TABLE 9.4: Statistics of the runtimes of first-order and higher-order program implementations, including the medians (m), means (μ), standard deviations (σ) and number of higher-order calls.

Due to the nature of the distribution of the runtimes of the programs not following normal distributions, the analysis will reference the median runtimes of each program implementation. While the mean and median are similar due to the relatively large

number of trials, the median forms a better representation of the data, and has been used in similar benchmarking studies [8].

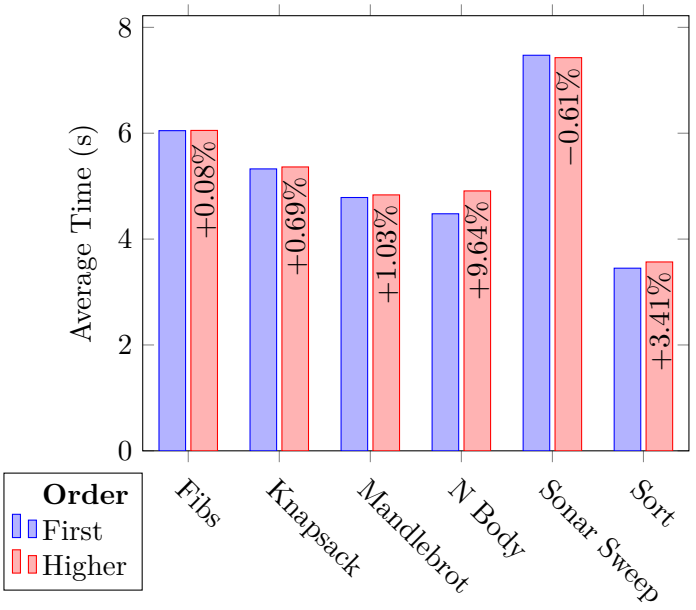


FIGURE 9.1: Median runtimes of first-order and higher-order program implementations, annotated with the relative change in the median runtime between first-order and higher-order implementations.

Figure 9.1 shows the relative difference in the runtimes of first-order and higher-order program implementations. In the case of the *Fibs* program, the number of higher-order calls is small. As such, we expect to not see an overall runtime difference in the program, with the results following this expectation.

Likewise, the *Knapsack* program performs a relatively small number of higher-order calls. Again, we see a negligible difference in the runtimes between the higher-order and first-order implementations.

The *Sonar Sweep* program performs a relatively higher number of higher-order calls. interestingly, however, the runtime of the higher-order implementation is lower than that of the first-order implementation. This difference is too small to conclude that the higher-order implementation is faster overall. Conversely, while the *Sort* program performs a similar number of higher-order calls, the complexity of these higher-order calls is greater, performing more comparisons in general than the *Sonar Sweep* program, which contributes to the increased runtime of the higher-order implementation.

The *Mandlebrot* program performs the greatest number of higher-order calls. While the runtime of the higher-order implementation of the program is slower, this slow down is

relatively small. The computation performed in the higher-order call is relatively simple, being floating-point operations and a conditional branch, which likely contributes to the relatively small overhead.

The *N Body* program performs the worst of all higher-order implementations compared to their first-order counterparts. The *N Body* program is the most complex of the test suite, performing a nested higher-order call, with a `fold` call inside a `map` call. The higher-order term generated in the nested higher-order call, `update_v(dt)`, must be allocated dynamically due to it closing over a variable value. This contributes to the deterioration of the higher-order implementation runtime, as this allocation occurs in each invocation of the higher-order term of the `map` procedure.

This introduces the possibility of a future optimisation within the Wybe compiler. As this higher-order term is an invariant of the anonymous procedure, the generation of the term could be hoisted outside of the procedure and passed in as a fresh closed variable to the mapped procedure.

9.4.3.2 Resource Implementation Strategy

Statistics of the execution runtime of the 30 trials of first-order implementations, when compiled using the parameterised and globalised resource implementation strategies, are shown in Table 9.5.

Program	Parameterised (s)			Globalised (s)		
	m	μ	σ	m	μ	σ
Fibs	6.464	6.463	0.017	6.043	6.045	0.013
Knapsack	5.337	5.324	0.055	5.307	5.299	0.070
Mandlebrot	5.022	5.054	0.102	4.909	4.935	0.079
N Body	4.526	4.545	0.055	4.492	4.513	0.076
Sonar Sweep	7.470	7.474	0.014	7.477	7.483	0.023
Sort	3.475	3.478	0.022	3.454	3.464	0.062

TABLE 9.5: Statistics of the runtimes of globalised and parameterised implementations, including the medians (m), means (μ), and standard deviations (σ).

Much like with the runtimes of the first-order and higher-order programs from before, due to the nature of the distributions of runtimes, we choose to analyse the median runtimes. Due to the relatively large number of trials the means and medians are close, however.

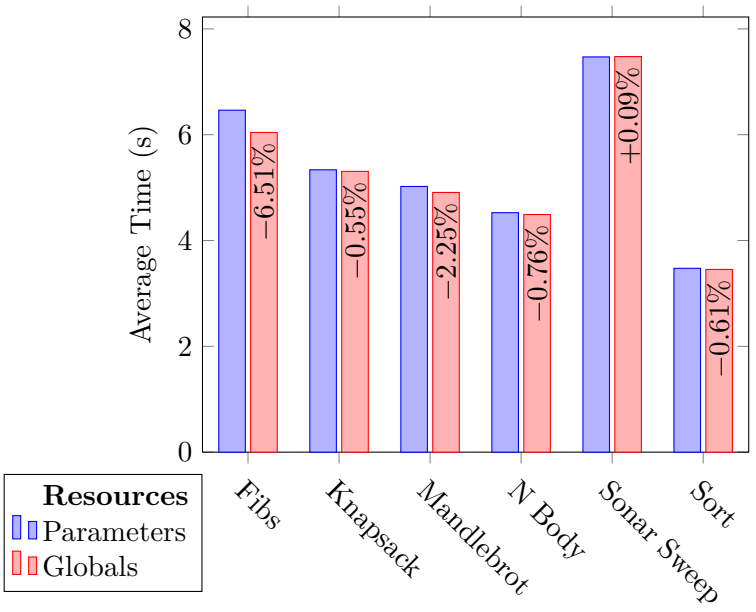


FIGURE 9.2: Median runtimes of programs with globalised and parameterised resources, annotated with the relative change in the median runtime between parameterised and globalised resources.

As shown in Figure 9.2, the relative difference in the median runtimes for the two resource implementations strategies are similar for the *Knapsack*, *Mandlebrot*, *N Body*, *Sonar Sweep*, and *Sort* programs. The *Fibs* program shows different behaviour, however.

In the case of the *Mandlebrot*, *N Body*, and *Sort* programs, the primary resource used is the `io` resource. As this resource has a *phantom-type*, in the translation into LLVM, all references to the resource are removed. This shows that, even with the globalised resource implementation, the `io` resource has zero overhead, continuing with that of the parameterised resource implementation strategy. These programs do see a slight decrease in runtime when compiled with globalised resources, however, the difference is small and within a small margin of error. This difference is likely due to I/O being a noisy operation, which is out of the control of the program and evident due to the relatively higher standard-deviations of these programs' runtimes.

The *Sonar Sweep* program uses a `counter` resource to tally results before printing them. Interestingly, this `counter` resource being stored in a global variable or a local variable has little impact on the overall runtime. As neither implementation strategy performs significantly faster, we can infer that the global variable for the `counter` has likely been stored in a cache or a register, allowing for fast access, much like in the case where the resource is passed as a parameter, being stored in a stack frame or in a register.

In the case of the *Knapsack* program, the performance is again virtually identical when using both parameterised and globalised resources. This is the result of the same effects seen in the *Sonar Sweep* program.

The *Fibs* program similarly makes use of a `counter` resource in tallying the sum of various Fibonacci numbers. However, an interesting optimisation occurs within the LLVM compiler for the globalised implementation that did not occur for the parameterised version. The tail-call optimisation can occur where a (recursive) call is made in the tail position, allowing the tail-call to be eliminated, being replaced with branch instructions. The Wybe compiler provides hints to the LLVM compiler where these tail calls occur, however, the LLVM compiler is not liable to perform such optimisations.

In the `fold_step` procedure of the *Fibs* program, the generated LPVM and LLVM code, represent the loop as a tail-recursive procedure. When using globalised resources, this procedure does not return a value, storing its result in the `counter` global variable, whereas with parameterised resources the `counter` variable is passed as an output. This inhibits the tail-call elimination optimisation from occurring and has the effect of producing slower code when using parameterised resources. This explains the performance gap between the two implementations

Chapter 10

Conclusion

10.1 Contribution

The contribution of this work can be broken into three parts: a higher-order extension to the Wybe language, a higher-order extension to the LPVM intermediate representation, and an extension to LPVM to support global variables.

In support of the higher-order extension to the Wybe language, we have formalised the type system in place in the Wybe language. This formalisation provides a basis upon which future extensions to the Wybe type system can use in the effort of a formal basis for such extensions.

The higher-order extension to the Wybe language has introduced syntactic extensions to the Wybe language, which allow for more expressive, higher-order code, to be manifested in the Wybe language.

The extension of the Wybe language with higher-order programming prompted an extension to LPVM to allow for higher-order programming. This included the extension of optimisations in place to incorporate the presence of higher-order terms and introduced a new optimisation that allowed the reduction of higher-order calls to first-order calls.

This higher-order extension to the Wybe language motivated the promotion of resources in the Wybe language into global variables from the previous implementation that reduced resources to formal parameters. The Wybe language relies on LPVM as its primary intermediate representation, and hence we have introduced global variables into the intermediate representation.

Global variables often inhibit optimisations in place in compilers. With the constraints placed on global variables through their origins as resources in Wybe, we have devised a declarative interface, *global flows*. Global flows allow for program analyses and transformations currently in place in LPVM to be correct in the presence of global variables. We have also implemented a novel optimisation that enables analysis of the manipulation of global variables, reducing the manipulation of global variables where possible. This optimisation would not be possible without the use of global flows.

We have also evaluated the performance of the extensions made to the Wybe language. We have shown that in general, higher-order code allows for only a small overhead in the runtime of programs compared to a first-order implementation. Further, the increased expressiveness enabled by higher-order code allows for the production of succinct, more general code that has can lead for reductions in the generated code's size. There are some cases where higher-order code can lead to an increased runtime of close to 10%, which we aim to focus in future work.

We have further shown that the transformation of resources into global variables does not affect the execution runtime nor the compiled program size greatly in general. There are some cases where the globalisation of resources allowed the LLVM compiler to produce a more optimised executable, decreasing the runtime with globalised resources, also.

10.2 Future Work

10.2.1 Multiple Specialisations

Within the Wybe compiler is a framework that supports *multiple specialisations* [3]. This framework allows different versions of procedures to exploit properties of the call sites of a procedure and is used currently to provide compile-time garbage collection in the Wybe language.

Specialisations of higher-order procedures, such as `map` or `sort` could be made where the higher-order argument (the mapping procedure and comparison procedure, respectively) are known. These specialisations would allow for other analyses to perform stronger optimisations, such as reducing the higher-order call to a first-order call, potentially bridging the performance gap between first-order and higher-order programming further.

This framework could also be used in future work to replace globalised resources with parameterised resources as was previously the case in the Wybe compiler. Although, the benchmarking experiments we have performed in this work have shown that there are negligible overheads associated with the use of global variables on the runtime of programs. Excessive multiple specialisations may introduce a small increase in the binary final size, and the performance gain may be negligible, so further work is required to determine if multiple specialisations with globalised resources would be fruitful.

10.2.2 Globalisation Limitation

In Section 5.2.2, we have discussed one major limitation to the globalisation of resources. This occurs in the presence of failing tests, where the globalised resource does not correctly revert the value to that of before the test. Future work may be able to devise an enhancement of the globalisation strategy to amend this limitation.

In logic programming, the concept of a *trail* is used to track the state of variables that are to be appropriately unbound in the case of failure, and this approach may be suitable in the correction of this globalisation strategy. Alternatively, a different strategy may appear that does not have this limitation.

10.2.3 Improved Closure Allocation

As shown with the higher-order implementation of the *N Body* program, identical closures may be constructed numerous times throughout a higher order call. If these closures are not constant, this requires allocation of memory on the heap, which we have shown introduces a slowdown in the execution runtime of the compiled code.

The compiler currently is unable to share references to these constructed closures as they occur in separate calls. However, the compiler may recognise that the term is defined as

an invariant of the closure, and transform the closure procedure, hoisting the invariant term as a fresh closed variable of the closure. This would further require transformation of all references to this closure to construct the term and pass it as a closed variable to the closure.

In future work we aim to hoist these common terms from closure procedures in the effort to bridge the gap in performance seen with higher-order code. This optimisation could also apply further to any procedure where a term is an invariant of the input parameters, and extract such terms from other procedures, such as the recursive procedures constructed to represent loops during unbranching.

Appendix A

Evaluation Data

Table A.1: Runtime of first-order and higher-order program implementations.

Program	Order	Time (s)							
Fibs	First	6.072	6.043	6.046	6.060	6.058	6.031	6.059	6.025
		6.076	6.039	6.048	6.038	6.037	6.032	6.083	6.033
		6.035	6.058	6.049	6.046	6.054	6.093	6.052	6.063
		6.040	6.036	6.035	6.054	6.059	6.062		
	Higher	6.060	6.092	6.039	6.045	6.079	6.060	6.045	6.034
		6.058	6.049	6.053	6.051	6.048	6.053	6.065	6.055
		6.051	6.066	6.055	6.041	6.029	6.060	6.061	6.045
		6.044	6.069	6.105	6.041	6.081	6.017		
Knapsack	First	5.282	5.273	5.345	5.318	5.275	5.299	5.301	5.343
		5.288	5.408	5.260	5.305	5.368	5.332	5.317	5.360
		5.475	5.361	5.325	5.324	5.265	5.371	5.366	5.241
		5.351	5.326	5.413	5.344	5.304	5.356		
	Higher	5.350	5.442	5.365	5.353	5.366	5.369	5.494	5.305
		5.457	5.316	5.307	5.268	5.294	5.413	5.368	5.369
		5.427	5.370	5.392	5.368	5.313	5.396	5.438	5.336
		5.304	5.279	5.351	5.361	5.347	5.318		
Mandlebrot	First	4.982	4.905	4.743	4.790	4.741	4.926	4.780	4.775
		4.801	5.083	4.736	4.773	4.748	4.771	4.784	5.121
		4.780	4.887	4.751	4.759	4.764	4.769	4.989	4.771
		4.805	4.824	4.896	4.798	4.787	4.822		
	Higher	4.940	4.805	4.827	4.837	4.809	4.838	4.832	4.893
		4.999	4.787	4.818	4.902	4.800	4.812	5.142	5.160
		4.979	4.789	4.959	4.892	4.864	4.797	4.792	4.778
		4.860	4.795	4.862	4.958	4.822	4.828		
N Body	First	4.530	4.573	4.453	4.477	4.493	4.553	4.477	4.471
		4.721	4.489	4.484	4.476	4.449	4.528	4.673	4.465
		4.477	4.476	4.493	4.526	4.436	4.479	4.486	4.468
		4.442	4.480	4.458	4.465	4.444	4.489		
	Higher	4.960	4.908	4.908	4.985	4.909	4.870	5.041	4.914
		4.957	4.935	4.933	4.989	4.929	4.889	4.890	4.865
		4.909	4.844	4.359	5.066	4.878	5.030	4.882	4.898
		4.877	5.022	4.928	4.910	4.900	4.920		
Sonar Sweep	First	7.470	7.478	7.488	7.468	7.470	7.551	7.487	7.490

TABLE A.1: Runtime of first-order and higher-order program implementations.

Program	Order	Time (s)							
		7.569	7.473	7.472	7.475	7.472	7.476	7.468	7.477
		7.470	7.472	7.469	7.474	7.473	7.691	7.472	7.473
		7.473	7.470	7.471	7.473	7.470	7.471		
	Higher	7.430	7.437	7.430	7.430	7.421	7.426	7.427	7.425
		7.430	7.435	7.425	7.423	7.426	7.432	7.432	7.426
		7.429	7.423	7.425	7.425	7.424	7.429	7.428	7.431
		7.423	7.429	7.427	7.422	7.428	7.425		
Sort	First	3.435	3.421	3.423	3.447	3.452	3.425	3.781	3.449
		3.414	3.465	3.423	3.486	3.473	3.416	3.425	3.412
		3.488	3.422	3.479	3.490	3.469	3.443	3.467	3.439
		3.746	3.435	3.499	3.462	3.471	3.470		
	Higher	3.557	3.562	3.536	3.561	3.561	3.576	3.554	3.571
		3.568	3.627	3.836	3.551	3.570	3.611	3.595	3.581
		3.596	3.546	3.565	3.555	3.579	3.574	3.542	3.572
		3.581	3.566	3.538	3.581	3.568	3.563		

TABLE A.2: Runtime of first-order programs with globalised and parameterised resources.

Program	Resource	Time (s)							
Fibs	Parameters	6.428	6.421	6.456	6.463	6.440	6.452	6.472	6.452
		6.430	6.473	6.459	6.473	6.464	6.474	6.488	6.461
		6.447	6.464	6.484	6.460	6.467	6.486	6.467	6.463
		6.492	6.457	6.481	6.472	6.471	6.463		
	Globals	6.040	6.062	6.053	6.046	6.021	6.039	6.041	6.031
		6.037	6.049	6.058	6.070	6.048	6.044	6.044	6.042
		6.032	6.053	6.066	6.048	6.040	6.048	6.042	6.033
		6.039	6.039	6.066	6.037	6.072	6.022		
Knapsack	Parameters	5.279	5.381	5.329	5.361	5.357	5.262	5.299	5.257
		5.335	5.370	5.199	5.393	5.210	5.324	5.381	5.384
		5.251	5.354	5.269	5.338	5.322	5.273	5.351	5.406
		5.310	5.359	5.348	5.383	5.349	5.274		
	Globals	5.329	5.354	5.357	5.077	5.191	5.190	5.363	5.218
		5.305	5.328	5.269	5.289	5.379	5.360	5.262	5.298
		5.330	5.361	5.259	5.254	5.301	5.234	5.385	5.288
		5.367	5.309	5.263	5.330	5.316	5.411		
Mandlebrot	Parameters	5.040	5.101	5.048	4.976	5.330	4.996	5.197	4.969
		5.103	5.139	4.988	4.979	5.022	4.952	4.919	5.006
		5.053	5.177	5.041	4.992	5.014	5.276	5.121	5.022
		4.965	5.174	4.960	4.956	4.960	5.136		
	Globals	4.994	4.930	4.951	4.901	4.953	5.086	5.064	4.899
		4.898	4.918	4.898	5.232	4.919	4.912	4.889	4.870
		4.884	4.920	4.893	4.929	4.913	4.887	4.880	5.060
		4.911	4.904	4.882	4.877	4.900	4.907		
N Body	Parameters	4.596	4.572	4.523	4.583	4.650	4.613	4.575	4.527
		4.565	4.502	4.550	4.491	4.579	4.510	4.512	4.751
		4.499	4.512	4.524	4.526	4.502	4.494	4.530	4.510
		4.571	4.516	4.500	4.531	4.514	4.540		
	Globals	4.523	4.539	4.534	4.541	4.550	4.579	4.488	4.463
		4.502	4.455	4.503	4.482	4.480	4.465	4.480	4.528
		4.459	4.546	4.466	4.723	4.498	4.491	4.441	4.462

TABLE A.2: Runtime of first-order programs with globalised and parameterised resources.

Program	Resource	Time (s)							
		4.582	4.773	4.458	4.424	4.459	4.493		
Sonar Sweep	Parameters	7.469	7.525	7.470	7.476	7.471	7.521	7.471	7.470
		7.471	7.468	7.481	7.471	7.475	7.471	7.475	7.474
		7.471	7.471	7.468	7.467	7.470	7.467	7.466	7.474
		7.468	7.469	7.464	7.469	7.470	7.470		
	Globals	7.471	7.473	7.479	7.474	7.474	7.472	7.482	7.478
		7.476	7.477	7.472	7.469	7.468	7.477	7.470	7.500
		7.495	7.481	7.480	7.598	7.492	7.484	7.495	7.480
		7.477	7.491	7.490	7.475	7.471	7.474		
Sort	Parameters	3.473	3.443	3.526	3.506	3.513	3.476	3.492	3.487
		3.469	3.446	3.481	3.480	3.470	3.455	3.497	3.465
		3.522	3.491	3.476	3.498	3.474	3.463	3.457	3.465
		3.462	3.489	3.463	3.472	3.439	3.485		
	Globals	3.484	3.440	3.425	3.426	3.465	3.454	3.475	3.428
		3.460	3.418	3.449	3.457	3.478	3.453	3.462	3.448
		3.769	3.468	3.446	3.444	3.428	3.435	3.486	3.512
		3.460	3.444	3.474	3.424	3.461	3.436		

Appendix B

Evaluation Programs

B.1 Fibs

The *Fibs* program naïvely calculates the sum of the first n Fibonacci numbers, for each input n, printing each result. The partial sums are stored in a resource.

This program has a high computational complexity, however, this complexity is not placed in the higher-order component of this program, and instead in the higher-order term the `range` is `fold`ed over.

```
1  resource counter:int
2
3  def fold_step(r:range) use !counter {
4      for ?i in r {
5          !step(i)
6      }
7  }
8
9  def step(i:int) use !counter {
10     !counter += fib(i)
11 }
12
13 def fib(n:int) = if { n <= 1 :: n | else :: fib(n - 1) + fib(n - 2) }
14
15 do {
16     !read(?n)
17     while n > 0
18
19     use counter in {
20         ?counter = 0
21         !fold_step(0..n)
22         !println(counter)
23     }
```

```
24 }
```

LISTING B.1: The first-order implementation of the Fibs program.

```
1  resource counter:int
2
3  def fold(f:{resource}(int), r:range) {
4      for ?i in r {
5          !f(i)
6      }
7  }
8
9  def step(i:int) use !counter {
10     !counter += fib(i)
11 }
12
13 def fib(n:int) = if { n <= 1 :: n | else :: fib(n - 1) + fib(n - 2) }
14
15 do {
16     !read(?n)
17     while n > 0
18
19     use counter in {
20         ?counter = 0
21         !fold(step, 0..n)
22         !println(counter)
23     }
24 }
```

LISTING B.2: The higher-order implementation of the Fibs program.

B.2 Knapsack

The *Knapsack* program is a program that solves the 0-1 knapsack problem using dynamic programming. The program reports the solution to the problem over an arbitrary collection of items read as input.

The program uses a resource, `sack`, to store the previous row calculated for the solution, and progressively builds the matrix, row by row until fully populated.

```
1  type item {
2      pub item(weight:int, value:int)
3  }
4
5  resource sack:list(int)
6
7  def init_sack(capacity:int) use ?sack {
8      if { capacity <= 0 ::
9          ?sack = [0]
10     | else ::
11         !init_sack(capacity - 1)
12         ?sack = [0 | sack]
13     }
14 }
15
16 def knapsack(capacity:int, item:item) use !sack {
17     !map_add_item(item, 0..(capacity + 1), ?sack)
18 }
19
20 def add_item(item:item, weight:int, ?cost:int) use sack {
21     ?cost = 0
22     if { sack[weight] = ?cost ::
23         if { sack[weight - item^weight] = ?prev_cost ::
24             ?cost = max(item^value + prev_cost, cost)
25         }
26     }
27 }
28
29 def map_knapsack(capacity:int, items:list(item)) use !sack {
30     if { items = [?item | ?items] ::
31         !knapsack(capacity, item)
32         !map_knapsack(capacity, items)
33     }
34 }
35
36 def map_add_item(item:item, r:range, ?values:list(int)) use sack {
37     ?values = []
38     for ?i in r {
39         !add_item(item, i, ?value)
40         ?values = [value | values]
41     }
42     ?values = reverse(values)
```

```
}

use sack in {
    !read(?capacity)
    !read(?n)

    ?items = []
    for _ in 0..n {
        !read(?weight)
        !read(?value)
        ?items = [item(weight, value) | items]
    }

    !init_sack(capacity)
    !map_knapsack(capacity, items)
    if { sack[capacity] = ?solution ::
        !println(solution)
    }
}
```

Listing B.3: The first-order implementation of the Knapsack program.

```
type item {
    pub item(weight:int, value:int)
}

resource sack:list(int)

def init_sack(capacity:int) use ?sack {
    if { capacity <= 0 ::
        ?sack = [0]
    | else ::
        !init_sack(capacity - 1)
        ?sack = [0 | sack]
    }
}

def knapsack(capacity:int, item:item) use !sack {
    !map(add_item(item), 0..(capacity + 1), ?sack)
}

def add_item(item:item, weight:int, ?cost:int) use sack {
    ?cost = 0
    if { sack[weight] = ?cost ::
        if { sack[weight - item^weight] = ?prev_cost ::
            ?cost = max(item^value + prev_cost, cost)
        }
    }
}

def map(f:{resource}(X), xs:list(X)) {
```

```
30      if { xs = [?x | ?xs] ::
31          !f(x)
32          !map(f, xs)
33      }
34  }
35
36  def map(f:{resource}(int, ?X), r:range, ?xs:list(X)) {
37      ?xs = []
38      for ?i in r {
39          !f(i, ?x)
40          ?xs = [x | xs]
41      }
42      ?xs = reverse(xs)
43  }
44
45  use sack in {
46      !read(?capacity)
47      !read(?n)
48
49      ?items = []
50      for _ in 0..n {
51          !read(?weight)
52          !read(?value)
53          ?items = [item(weight, value) | items]
54      }
55
56      !init_sack(capacity)
57      !map(knapsack(capacity), items)
58      if { sack[capacity] = ?solution ::
59          !println(solution)
60      }
61  }
```

Listing B.4: The higher-order implementation of the Knapsack program.

B.3 Mandlebrot

The *Mandlebrot* program is a program that prints the Mandlebrot set, an object from mathematics that is the set of complex numbers. A complex number, c, is in the Mandlebrot set if and only if the critical point, 0, under iteration of the quadratic map $z_{n+1} = z_n^2 + c$ is bounded. We approximate the Mandlebrot set by iterating a fixed number of times.

This program again has a high computational complexity, however, the computational complexity is due to a large number of iterations the quadratic map performs. The actual map itself is relatively computationally simple.

```
1  type complex {
2      pub `~`(i:float, j:float)
3
4      pub def (x:_ + y:_):_ = x^i + y^i ~ x^j + y^j
5
6      pub def (x:_ * y:_):_ = x^i * y^i - x^j * y^j ~ x^i * y^j + x^j * y^i
7
8      pub def square_magnitude(c:_):float = c^i * c^i + c^j * c^j
9  }
10
11 def {test} iter(c:complex, !z:complex) {
12     ?z = z * z + c
13     square_magnitude(z) <= 4.0
14 }
15
16 def {test} repeat_iter(n:int, c:complex, x:complex) {
17     if { n > 0 ::
18         iter(c, !x)
19         repeat_iter(n - 1, c, x)
20     }
21 }
22
23 !read(?n)
24 !read(?d)
25
26 ?j = -2.0
27 do {
28     while j < 2.0
29     ?i = -2.0
30     do {
31         while i < 2.0
32         ?c = (i ~ j)
33         repeat_iter(n, c, 0.0 ~ 0.0, ?in_set)
34         !print(if { in_set :: "* " | else :: "  " })
35         !i += d
36     }
37     !j += d
38     !nl
```

```
39 }
```

Listing B.5: The first-order implementation of the Mandlebrot program.

```
1  type complex {
2      pub `~`(i:float, j:float)
3
4      pub def (x:_ + y:_):_ = x^i + y^i ~ x^j + y^j
5
6      pub def (x:_ * y:_):_ = x^i * y^i - x^j * y^j ~ x^i * y^j + x^j * y^i
7
8      pub def square_magnitude(c:_):float = c^i * c^i + c^j * c^j
9  }
10
11 def {test} iter(c:complex, !z:complex) {
12     ?z = z * z + c
13     square_magnitude(z) <= 4.0
14 }
15
16 def {test} repeat(n:int, p:{test}(!X), x:X) {
17     if { n > 0 ::
18         p(!x)
19         repeat(n - 1, p, x)
20     }
21 }
22
23 !read(?n)
24 !read(?d)
25
26 ?j = -2.0
27 do {
28     while j < 2.0
29     ?i = -2.0
30     do {
31         while i < 2.0
32         ?c = (i ~ j)
33         repeat(n, {test}{ iter(c, !@) }, 0.0 ~ 0.0, ?in_set)
34         !print(if { in_set :: "* " | else :: "  " })
35         !i += d
36     }
37     !j += d
38     !nl
39 }
```

Listing B.6: The higher-order implementation of the Mandlebrot program.

B.4 N Body

The *N Body* program is a program used to simulate the N body problem. The N body problem is a problem from computational physics and involves the simulation of a system of bodies in a gravitational space. Due to the complex nature of the system in simulation, the program approximates the system in discretised time steps. The program simulates this problem over a finite number of time steps on a subset of the planets in our solar system, orbiting the Sun.

This program has a high computational complexity. However, in this program, there is a balance between the number of higher-order calls performed and the complexity of the higher-order terms.

```
1  type body {
2      pub body(x:float, y:float, vx:float, vy:float, m:float)
3
4      pub def print(body:_) use !io {
5          !print("(")
6          !print(body^x)
7          !print(", ")
8          !print(body^y)
9          !print("); ")
10     }
11 }
12
13 def map_print(bodies:list(body)) use !io {
14     if { bodies = [?body | ?rest] ::
15         !print(body)
16         !map_print(rest)
17     }
18 }
19
20 def step(dt:float, !bodies:list(body)) {
21     map_update_v(dt, bodies, !bodies)
22     map_update_pos(dt, !bodies)
23 }
24
25 def map_update_v(dt:float, all:list(body), !bodies:list(body)) {
26     if { bodies = [?body | ?rest] ::
27         fold_update_v(dt, all, !body)
28         map_update_v(dt, all, !rest)
29         ?bodies = [body | rest]
30     }
31 }
32
33 def fold_update_v(dt:float, all:list(body), !body:body) {
34     if { all = [?other | ?all] ::
35         update_v(dt, other, !body)
36         fold_update_v(dt, all, !body)
37     }
38 }
```

```
39
40 def update_v(dt:float, other:body, !body:body) {
41     body(?x1, ?y1, ?vx, ?vy, ?m1) = body
42     body(?x2, ?y2, _, _, ?m2) = other
43     ?dx = x1 - x2
44     ?dy = y1 - y2
45     ?d2 = dx * dx + dy * dy
46     if { d2 ~= 0.0 ::
47         ?mag = m2 * dt * (d2 ** -1.5)
48         ?body = body(x1, x2, vx - dx * mag, vy - dy * mag, m1)
49     }
50 }
51
52 def map_update_pos(dt:float, !bodies:list(body)) {
53     if { bodies = [?body | ?rest] ::
54         update_pos(dt, !body)
55         map_update_pos(dt, !rest)
56         ?bodies = [body | rest]
57     }
58 }
59
60 def update_pos(dt:float, !body:body) {
61     body(?x, ?y, ?vy, ?vx, ?m) = body
62     ?body = body(x + dt * vx, y + dt * vy, vx, vy, m)
63 }
64
65 ?bodies = [
66     body(15.37969711485, -25.91931460998, # Neptune
67          0.97909073224, 0.59469899864, 0.00203368686),
68     body(12.89436956213, -15.11115140169, # Uranus
69          1.08279100644, 0.86871301817, 0.00172372405),
70     body(8.34336671824, 4.12479856412, # Saturn
71          -1.01077434618, 1.82566237123, 0.01128632613),
72     body(4.84143144246, -1.16032004402, # Jupiter
73          0.60632639299, -0.02521836165, 0.03769367487),
74     body(0.0, 0.0, 0.0, 0.0, 39.4784176044) # Sun
75 ]
76
77 !read(?n)
78 for _ in 0..n {
79     step(0.01, !bodies)
80 }
81 !map_print(bodies)
```

Listing B.7: The first-order implementation of the N Body program.

```
1 type body {
2     pub body(x:float, y:float, vx:float, vy:float, m:float)
3
4     pub def print(body:_) use !io {
5         !print("(")
```

```
6           !print(body^x)
7           !print(", ")
8           !print(body^y)
9           !print("); ")
10      }
11  }
12
13  def map(f:{resource}(X), xs:list(X)) {
14      if { xs = [?x | ?xs] ::
15          !f(x)
16          !map(f, xs)
17      }
18  }
19
20  def map(f:(!X), !xs:list(X)) {
21      if { xs = [?x | ?rest] ::
22          f(!x)
23          map(f, !rest)
24          ?xs = [x | rest]
25      }
26  }
27
28  def fold(f:(X, !S), xs:list(X), !s:S) {
29      if { xs = [?x | ?rest] ::
30          f(x, !s)
31          fold(f, rest, !s)
32      }
33  }
34
35  def step(dt:float, !bodies:list(body)) {
36      map({ fold(update_v(dt), bodies, !@) }, !bodies)
37      map({ update_pos(dt, !@) }, !bodies)
38  }
39
40  def update_v(dt:float, other:body, !body:body) {
41      body(?x1, ?y1, ?vx, ?vy, ?m1) = body
42      body(?x2, ?y2, _, _, ?m2) = other
43      ?dx = x1 - x2
44      ?dy = y1 - y2
45      ?d2 = dx * dx + dy * dy
46      if { d2 ~= 0.0 ::
47          ?mag = m2 * dt * (d2 ** -1.5)
48          ?body = body(x1, x2, vx - dx * mag, vy - dy * mag, m1)
49      }
50  }
51
52  def update_pos(dt:float, !body:body) {
53      body(?x, ?y, ?vx, ?vy, ?m) = body
54      ?body = body(x + dt * vx, y + dt * vy, vx, vy, m)
55  }
56
57  ?bodies = [
```

```
58      body(15.37969711485, -25.91931460998, # Neptune
59           0.97909073224, 0.59469899864, 0.00203368686),
60      body(12.894369562139, -15.11115140169, # Uranus
61           1.08279100644, 0.86871301817, 0.00172372405),
62      body(8.34336671824, 4.12479856412, # Saturn
63           -1.01077434618, 1.82566237123, 0.01128632613),
64      body(4.84143144246, -1.16032004402, # Jupiter
65           0.60632639299, -0.02521836165, 0.03769367487),
66      body(0.0, 0.0, 0.0, 0.0, 39.4784176044) # Sun
67  ]
68
69  !read(?n)
70  for _ in 0..n {
71      step(0.01, !bodies)
72  }
73  !map(body.print, bodies)
```

Listing B.8: The higher-order implementation of the N Body program.

B.5 Sonar Sweep

The *Sonar Sweep* program is a program used to solve the problem of day 1 of the 2021 Advent of Code[1]. This program is given a list of integer depths and outputs the number of depths that when considered in successive pairs increase, generalised beyond that of the problem in the Advent of Code to an arbitrary distance between pairs of depths.

This program makes use of a resource to keep a count of the number of increasing depths that have been seen, updating this count many times.

```
1  resource counter:int
2
3  def read_ints(?is:list(int)) use !io {
4      ?is = []
5      do {
6          !read(?i)
7          until i < 0
8          ?is = [i | is]
9      }
10     reverse(is, ?is)
11 }
12
13 def foldn_incr_increasing(n:int, as:list(int)) use !counter {
14     if { as = [?a0 | ?as] & as[n] = ?an ::
15         !incr_increasing(a0, an)
16         !foldn_incr_increasing(n, as)
17     }
18 }
19
20 def incr_increasing(a:int, b:int) use !counter {
21     if { a < b :: incr(!counter) }
22 }
23
24 def solve(n:int, depths:list(int), ?ans:int) {
25     use counter in {
26         ?counter = 0
27         !foldn_incr_increasing(n, depths)
28         ?ans = counter
29     }
30 }
31
32 !read(?n)
33 !read_ints(?depths)
34 for ?i in 0..n {
35     !println(solve(i, depths))
36 }
```

Listing B.9: The first-order implementation of the Sonar Sweep program.

[1]The full problem can be found here: https://adventofcode.com/2021/day/1

```
resource counter:int

def read_ints(?is:list(int)) use !io {
    ?is = []
    do {
        !read(?i)
        until i < 0
        ?is = [i | is]
    }
    reverse(is, ?is)
}

def fold_window(n:int, f:{resource}(A,A), as:list(A)) {
    if { as = [?a0 | ?as] & as[n] = ?an ::
        !f(a0, an)
        !fold_window(n, f, as)
    }
}

def incr_if_increasing(a:int, b:int) use !counter {
    if { a < b :: incr(!counter) }
}

def solve(n:int, depths:list(int), ?ans:int) {
    use counter in {
        ?counter = 0
        !fold_window(n, incr_if_increasing, depths)
        ?ans = counter
    }
}

!read(?n)
!read_ints(?depths)
for ?i in 0..n {
    !println(solve(i, depths))
}
```

Listing B.10: The higher-order implementation of the Sonar Sweep program.

B.6 Sort

The *Sort* program implements a sorting procedure, defined over various types. In the first-order implementation, the sort program must be duplicated, as there is no mechanism to factor out the common component (‘<=‘, the procedure used to order values). This contrasts the higher-order implementation where the comparison predicate is a parameter of the `sort` and `merge` predicates.

The sorting algorithm chosen, the insertion sort, was chosen due to the relative inefficiency of the sorting algorithm. The purpose of the program is not to optimally sort the given input, however, to contrast the effects of higher-order code. As an insertion sort for an input of size N performs $O(N^2)$ comparisons, this will perform a greater number of higher-order calls than that of a more efficient sort that may perform $O(N \log N)$ comparisons.

```
1  module ints {
2      def sort(xs:list(int), ?sorted:list(int)) {
3          ?sorted = []
4          for ?x in xs {
5              insert(x, !sorted)
6          }
7      }
8
9      def insert(x:int, !xs:list(int)) {
10         if { [?head | ?rest] = xs ::
11             if { x <= head ::
12                 ?xs = [x | xs]
13             | else ::
14                 insert(x, !rest)
15                 ?xs = [head | rest]
16             }
17         | else ::
18             ?xs = [x]
19         }
20     }
21
22     pub def read_sort_print(n:int) use !io {
23         ?xs = []
24         for ?i in 0..n {
25             !read(?x)
26             ?xs = [x | xs]
27         }
28         for ?x in sort(xs) {
29             !print(x)
30             !print(", ")
31         }
32         !nl
33     }
34 }
35
36 module floats {
```

```
37      def sort(xs:list(float), ?sorted:list(float)) {
38          ?sorted = []
39          for ?x in xs {
40              insert(x, !sorted)
41          }
42      }
43
44      def insert(x:float, !xs:list(float)) {
45          if { [?head | ?rest] = xs ::
46              if { x <= head ::
47                  ?xs = [x | xs]
48              | else ::
49                  insert(x, !rest)
50                  ?xs = [head | rest]
51              }
52          | else ::
53              ?xs = [x]
54          }
55      }
56
57      pub def read_sort_print(n:int) use !io {
58          ?xs = []
59          for ?i in 0..n {
60              !read(?x)
61              ?xs = [x | xs]
62          }
63          for ?x in sort(xs) {
64              !print(x)
65              !print(", ")
66          }
67          !nl
68      }
69  }
70
71  module strings {
72      def sort(xs:list(string), ?sorted:list(string)) {
73          ?sorted = []
74          for ?x in xs {
75              insert(x, !sorted)
76          }
77      }
78
79      def insert(x:string, !xs:list(string)) {
80          if { [?head | ?rest] = xs ::
81              if { x <= head ::
82                  ?xs = [x | xs]
83              | else ::
84                  insert(x, !rest)
85                  ?xs = [head | rest]
86              }
87          | else ::
88              ?xs = [x]
```

```
89          }
90      }
91
92      pub def read_sort_print(n:int) use !io {
93          ?xs = []
94          for ?i in 0..n {
95              !read(?x)
96              ?xs = [x | xs]
97          }
98          for ?x in sort(xs) {
99              !print(x)
100             !print(", ")
101         }
102         !nl
103     }
104 }
105
106 module chars {
107     def sort(xs:list(char), ?sorted:list(char)) {
108         ?sorted = []
109         for ?x in xs {
110             insert(x, !sorted)
111         }
112     }
113
114     def insert(x:char, !xs:list(char)) {
115         if { [?head | ?rest] = xs ::
116             if { x <= head ::
117                 ?xs = [x | xs]
118             | else ::
119                 insert(x, !rest)
120                 ?xs = [head | rest]
121             }
122         | else ::
123             ?xs = [x]
124         }
125     }
126
127     pub def read_sort_print(n:int) use !io {
128         ?xs = []
129         for ?i in 0..n {
130             !read(?x)
131             ?xs = [x | xs]
132         }
133         for ?x in sort(xs) {
134             !print(x)
135             !print(", ")
136         }
137         !nl
138     }
139 }
140
```

```
!read(?n)
!read(_:string)
!chars.read_sort_print(n)
!strings.read_sort_print(n)
!floats.read_sort_print(n)
!ints.read_sort_print(n)
```

LISTING B.11: The first-order implementation of the Sort program.

```
def sort(`<=`:{test}(X, X), xs:list(X), ?sorted:list(X)) {
    ?sorted = []
    for ?x in xs {
        insert(`<=`, x, !sorted)
    }
}

def insert(`<=`:{test}(X, X), x:X, !xs:list(X)) {
    if { [?head | ?rest] = xs ::
        if { x <= head ::
            ?xs = [x | xs]
        | else ::
            insert(`<=`, x, !rest)
            ?xs = [head | rest]
        }
    | else ::
        ?xs = [x]
    }
}

def read_sort_print(n:int, reader:{resource}(?X),
                    `<=`:{test}(X, X), printer:{resource}(X)) use !io {
    ?xs = []
    for ?i in 0..n {
        !reader(?x)
        ?xs = [x | xs]
    }
    for ?x in sort(`<=`, xs) {
        !printer(x)
        !print(", ")
    }
    !nl
}

!read(?n)
!read(_:string)
!read_sort_print(n, read, char.`<=`, print)
!read_sort_print(n, read, string.`<=`, print)
!read_sort_print(n, read, float.`<=`, print)
!read_sort_print(n, read, int.`<=`, print)
```

LISTING B.12: The higher-order implementation of the Sort program.

Bibliography

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

[2] Donald E. Knuth. The complexity of songs. *Commun. ACM*, 27(4):344–346, apr 1984. ISSN 0001-0782. doi: 10.1145/358027.358042. URL https://doi.org/10.1145/358027.358042.

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

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

[5] Ashutosh Rishi Ranjan. An incremental and work-saving compiler for the wybe programming langauge. Master's thesis, The University of Melbourne - School of Computing and Information Systems, 2016.

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

[7] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 190–203, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg. ISBN 978-3-540-39677-2.

[8] 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.

[9] 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.

[10] Peter J Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[11] Guy L Steele Jr. Rabbit: A compiler for scheme. Technical report, MIT Artificial Intelligence Laboratory Technical Report, 1978.

[12] Erik Sandewall. A proposed solution to the funarg problem. *SIGSAM Bull.*, 17(1):29–42, January 1971. ISSN 0163-5824. doi: 10.1145/1093420.1093422. URL https://doi.org/10.1145/1093420.1093422.

[13] Joel Moses. The function of function in lisp or why the funarg problem should be called the environment problem. *SIGSAM Bull.*, 15(1):13–27, jul 1970. ISSN 0163-5824. doi: 10.1145/1093410.1093411. URL https://doi.org/10.1145/1093410.1093411.

[14] Andrew W Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[15] Andrew W Appel and Zhong Shao. Empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47–74, 1996.

[16] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of standard ML: revised*. MIT press, 1997.

[17] 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.

[18] 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.

[19] 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, pages 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.

[20] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

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

[22] Henk Barendregt. Introduction to generalized type systems. *Journal of functional programming*, 1 (2):125–154, 1991.

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

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

[25] Joe B Wells. Typability and type checking in the second-order/spl lambda/-calculus are equivalent and undecidable. In *Proceedings Ninth Annual IEEE Symposium on Logic In Computer Science*, pages 176–185. IEEE, 1994.

[26] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

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

[28] Luis Damas. *Type assignment in programming languages*. PhD thesis, University of Edinburgh, 1984.

[29] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, 1989.

[30] Luca Cardelli, Simone Martini, John C Mitchell, and Andre Scedrov. An extension of system f with subtyping. *Information and computation*, 109(1-2):4–56, 1994.

[31] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for prolog. *Artificial Intelligence*, 23(3):295–307, 1984. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(84)90017-1. URL https://www.sciencedirect.com/science/article/pii/0004370284900171.

[32] T. Lakshman and Uday Reddy. Typed prolog: A semantic reconstruction of the mycroft-o'keefe type system. In *1991 International Logic Programming Symposium*, pages 202–217. MIT Press, 01 1991.

[33] G NADATHUR. An overview of λ prolog. In *5th Int. Conf. on Logic Programming*. MIT Press, 1988.

[34] 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.

[35] Roland Dietrich and Frank Hagl. A polymorphic type system with subtypes for prolog. In *European Symposium on Programming*, pages 79–93. Springer, 1988.

[36] Fred Chow. Intermediate representation: The increasing significance of intermediate representations in compilers. *Queue*, 11(10):30–37, oct 2013. ISSN 1542-7730. doi: 10.1145/2542661.2544374. URL https://doi.org/10.1145/2542661.2544374.

[37] 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, pages 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.

[38] 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.

[39] Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pages 133–138, New York, NY, USA, 1959. Association for Computing Machinery. ISBN 9781450378680. doi: 10.1145/1460299.1460314. URL https://doi.org/10.1145/1460299.1460314.

[40] Edward S. Lowry and C. W. Medlock. Object code optimization. *Commun. ACM*, 12(1):13–22, jan 1969. ISSN 0001-0782. doi: 10.1145/362835.362838. URL https://doi.org/10.1145/362835.362838.

[41] 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.

[42] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.

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

[44] 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.

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

[46] 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, pages 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.

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

[48] 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.

[49] 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.

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

[51] Alain Colmerauer and Philippe Roussel. The birth of prolog. *SIGPLAN Not.*, 28(3):37–52, mar 1993. ISSN 0362-1340. doi: 10.1145/155360.155362. URL https://doi.org/10.1145/155360.155362.

[52] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, jan 1965. ISSN 0004-5411. doi: 10.1145/321250.321253. URL https://doi.org/10.1145/321250.321253.

[53] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, page 207–212, New York, NY, USA, 1982. Association for Computing Machinery. ISBN 0897910656. doi: 10.1145/582153.582176. URL https://doi.org/10.1145/582153.582176.

[54] Karen Appleby, Mats Carlsson, Seif Haridi, and Dan Sahlin. Garbarge collection for prolog based on wam. *Commun. ACM*, 31(6):719–741, jun 1988. ISSN 0001-0782. doi: 10.1145/62959.62968. URL https://doi.org/10.1145/62959.62968.

[55] 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, pages 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.

[56] John Cocke. Global common subexpression elimination. In *Proceedings of a Symposium on Compiler Optimization*, pages 20–24, New York, NY, USA, 1970. Association for Computing Machinery. ISBN 9781450373869. doi: 10.1145/800028.808480. URL https://doi.org/10.1145/800028.808480.

[57] David A. Terei and Manuel M.T. Chakravarty. An llvm backend for ghc. *SIGPLAN Not.*, 45(11):109–120, sep 2010. ISSN 0362-1340. doi: 10.1145/2088456.1863538. URL https://doi.org/10.1145/2088456.1863538.

[58] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-language compiler benchmarking: Are we fast yet? *SIGPLAN Not.*, 52(2):120–131, nov 2016. ISSN 0362-1340. doi: 10.1145/3093334.2989232. URL https://doi.org/10.1145/3093334.2989232.

[59] Alvin Huseinović and Samir Ribić. Benchmark comparison of computing the mandelbrot set in opencl. In *2015 23rd Telecommunications Forum Telfor (TELFOR)*, pages 994–997, 2015. doi: 10.1109/TELFOR.2015.7377632.

[60] Sviatoslav Pestov, Daniel Ehrenberg, and Joe Groff. Factor: A dynamic stack-based programming language. *Acm Sigplan Notices*, 45(12):43–58, 2010.

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

[62] Iulian Dragoş. Optimizing higher-order functions in scala. In *Third International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, 2008. URL http://infoscience.epfl.ch/record/128135.