Phoebe Nichols

An Implementation of Prolog

Computer Science Tripos – Part II

Churchill College

2019

Declaration of Originality

I, Phoebe Nichols of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Phoebe Nichols of Churchill College, am content for my dissertation to be made available to the students and staff of the University.

Signed

Date

Acknowledgements

Many thanks to:

- My supervisor, **Prof. Alan Mycroft**, for his guidance and feedback
- **Matthew Ireland**, for his advice on this dissertation
- Friends and family, for proofreading

Proforma

Candidate number:	2398E
Project Title:	An Implementation of Prolog
Examination:	Computer Science Tripos – Part II, 2019
Word Count:	11848[1]
Lines of Code:	8384[2]
Page Count:	40
Project Originator:	Candidate 2398E
Supervisor:	Professor Alan Mycroft

Original aims of the Project

The project aimed to implement a compilation and execution system for the core features of Prolog in OCaml, exploring the advantages and disadvantages of OCaml over the more traditional choice of C. The core goals for the Prolog implementation included compiling to an abstract instruction set and writing an interpreter for this instruction set. Stretch goals for the project were implementing last-call optimisation (a generalised form of tail-recursion optimisation); adding type and mode systems; and performing an analysis to detect clauses that do not backtrack.

Work completed

A Prolog system (*Pholog*) was implemented, satisfying all of the core goals for the project and including significant extensions: Pholog includes a type system and performs last-call optimisation. Key advantages of Pholog over traditional Prolog implementations include exploiting garbage collection to simplify the run-time stack layout and the ability to catch certain programmer errors with its type system. In the evaluation section, I show that Pholog achieves comparable performance to SWI-Prolog (a popular C-based Prolog system), despite Pholog being a relatively small project and switching from C to OCaml.

Special difficulties

None.

[1] Obtained using `texcount -sum` from https://app.uio.no/ifi/texcount/

[2] Lines of OCaml, obtained using `cloc` from https://github.com/AlDanial/cloc

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
  - 1.1.1 Logic programming . . . 1
- 1.2 Project summary . . . 2
- 1.3 Previous work . . . 2

2 Preparation **3**
- 2.1 Programming in Prolog . . . 3
  - 2.1.1 Structure of a Prolog program . . . 3
- 2.2 Execution model for Prolog . . . 4
  - 2.2.1 Unification . . . 4
  - 2.2.2 Linear resolution . . . 4
  - 2.2.3 Search strategy . . . 5
  - 2.2.4 Backtracking . . . 5
  - 2.2.5 Cut . . . 5
  - 2.2.6 Negation . . . 5
- 2.3 The Warren Abstract Machine . . . 5
  - 2.3.1 Variable representation . . . 6
  - 2.3.2 Instructions . . . 6
  - 2.3.3 Memory areas . . . 6
- 2.4 The Mycroft-O'Keefe type system . . . 6
- 2.5 Choice of tools . . . 7
  - 2.5.1 OCaml . . . 7
- 2.6 Licensing . . . 9
- 2.7 Requirements analysis . . . 9
- 2.8 Starting point . . . 10
- 2.9 Development methodology . . . 10
  - 2.9.1 Testing strategy . . . 10
- 2.10 Summary . . . 12

3 Implementation **13**
- 3.1 Lexer and parser . . . 13
- 3.2 Abstract machine architecture . . . 13
  - 3.2.1 Term representation . . . 13
  - 3.2.2 Memory layout of the Pholog interpreter . . . 15
  - 3.2.3 Cut . . . 18
  - 3.2.4 Driver function for the abstract machine . . . 18
- 3.3 Instruction set . . . 18
  - 3.3.1 Last-call optimisation . . . 18
  - 3.3.2 Cut . . . 20
  - 3.3.3 Arithmetic . . . 20
- 3.4 Code generation . . . 21
  - 3.4.1 Translating variables to locations . . . 22
  - 3.4.2 Generating the instructions for each clause . . . 22
  - 3.4.3 Generating the instruction sequence . . . 24
  - 3.4.4 Summary of code generation . . . 24
  - 3.4.5 Relationship to the Warren Abstract Machine . . . 24

3.5 Type-checking . . . 26
3.5.1 Enforcing definitional generality . . . 27
3.6 Repository overview . . . 27
3.6.1 Module structure for the execute library . . . 28
3.6.2 Testing . . . 28
3.7 Summary . . . 30

4 Evaluation **31**
4.1 Success criterion . . . 31
4.1.1 Implementation correctness . . . 31
4.2 Test environment . . . 31
4.3 Speed of execution for benchmarks . . . 31
4.3.1 Benchmarks in which SWI-Prolog performs poorly . . . 32
4.4 Performance of garbage collection . . . 33
4.5 Last-call optimisation . . . 35
4.6 Type system . . . 36
4.6.1 Evaluation of correctness . . . 36
4.6.2 Errors that the type system can catch . . . 37
4.7 Comparison of implementations . . . 38
4.8 Summary . . . 39

5 Conclusions **40**
5.1 Work completed . . . 40
5.2 Lessons learned . . . 40
5.3 Future work . . . 40

Bibliography **Bib.1**

A Example Program **A.1**

B Extract from the Interpreter **B.1**

C Project Proposal **C.1**

List of Figures

2.1 Example showing the equivalence between Prolog clauses and Horn clauses. . . . 3
2.2 Memory layout of the WAM. . . . 6
2.4 Formal specification of the Mycroft-O'Keefe type system. . . . 8

3.1 High-level Pholog pipeline. . . . 13
3.2 Comparison between the memory layout used my interpreter and by the WAM. 14
3.3 Representation of the term `pair(one,Y)`. . . . 14
3.4 The aliasing of variables on unification. . . . 15
3.5 Creation of a pointer chain of length three. . . . 15
3.6 Diagram showing how pointers into the environment stack are used to implement backtracking. . . . 17
3.9 The effect of the instruction `GetStructArg g/3 <- Arg 0` when `Arg 0` contains an unbound variable. . . . 23
3.10 An explanation of the instructions generated for an example program. . . . 25
3.11 An example of the first pass of the type-checker, where variable types are inferred. `Autogen` types are used for the unification of unnamed type variables. . . . 27
3.12 An example showing how the first type-checker pass can infer a structure type. . 28
3.13 An example of type-checking that requires type variable bindings to be tracked whilst predicate calls are type-checked. . . . 28
3.14 Project dependancies. . . . 29

4.1 A comparison of the time taken to execute sample Prolog programs. The error bars show $\pm 5\sigma$. . . . 32
4.2 Scaling of the `test` predicate from Listing 4.1; the x-axis is the value of x. The line connecting points for SWI-Prolog uses quadratic interpolation. . . . 33
4.3 The effect of minor and major heap size on the execution time of the Ackermann benchmark. . . . 34
4.4 A comparison of the memory profile of the program in Listing 4.2 with and without last-call optimisation. . . . 35
4.5 A typed Prolog program, and the types inferred for its variables. . . . 36

List of Tables

2.1 Justification of project dependencies. . . . 9
2.2 Project implementation timeline. . . . 11

3.1 Summary of the instruction set. . . . 19

4.1 Test cases to demonstrate definitional generality of the type system. . . . 37
4.2 Comparison to existing Prolog implementations. . . . 38

List of Listings

2.1 Comparison between an append function implemented in ML and using the Mycroft-O'Keefe type system. . . . 7

3.1 Prolog program using arithmetic inside an `is` statement. 21
3.2 Instructions generated for example arithmetic expressions. 21
3.3 A clause before and after the abstraction of its variables. 22
3.4 Instructions to load the arguments of `cl(f(X,g(1,2),3))`. 23
3.5 Instructions to put values into the argument array for the predicate call `cl(f(X,g(1,2),3), Y, 4)`. 24
4.1 Example program to show the poor scaling of SWI-Prolog when allocating structures with a large stack. The value of x is varied in Figure 4.2. 33
4.2 A Prolog program expected to benefit significantly from last-call optimisation. . 35
4.3 Typed Prolog program using types defined in terms of other types. 37

Chapter 1

Introduction

This dissertation presents *Pholog*, an implementation of Prolog in a language with automatic memory management. Pholog is a compiler and abstract machine for Prolog written in OCaml, whereas traditional Prolog systems use C. I explore the advantages and disadvantages of this switch, and its implications for the design of a Prolog interpreter.

1.1 Motivation

The project is motivated by improvements over existing Prolog systems:

– **Simplification of run-time memory management**

  Implementing a Prolog interpreter is complex, partly due to the problem of run-time memory management. One difficulty associated with this memory management is implementing *environment protection*: Prolog implementations typically store *environments* and *choice points* in the same stack, and environment protection ensures that an environment is not removed whilst it is needed for a choice point.

  Pholog exploits the OCaml garbage collector to simplify its memory layout, avoiding concern with environment protection. Pholog uses two separate stacks for envirnonments and choice points, and relies on the garbage collector to detect when it is safe to remove an environment. This significantly reduces the chance of memory allocation errors in Pholog, and means that *last-call optimisation* is simple to implement.

– **Static error detection**

  Prolog programs are error-prone. Typically, the only classes of errors statically detected by Prolog systems are syntax errors and calling *predicates* (analogous to functions) with the incorrect name.

  Pholog includes a polymorphic type system. Adding a static type system enables a wider class of errors to be detected, such as mistyped *terms* and *predicate arguments* given in the wrong order.

– **Choice of implementation language**

  Prolog systems are typically implemented in C, including the popular implementations SWI-Prolog and GNU-Prolog. C lacks strong typing and memory safety, increasing the likelihood of errors in C-based Prolog implementations.

  Pholog uses OCaml for compiling and interpreting Prolog, which includes the features missing from C described above.

In my evaluation, I compare Pholog to SWI-Prolog and show that the benefits of Pholog do not come at a significant cost to performance. Pholog is also relatively complete: all the core features of Prolog have been implemented, as well as language extensions such as arithmetic and cut.

1.1.1 Logic programming

Prolog is a logic programming language; logic programming is a paradigm in which programs specify a series of logical statements that are used to derive the program's result. In contrast to logic programming languages, imperative languages consist of a sequence of commands for the computer to perform. This requires the programmer to understand the capabilities of the

computer, at some level. Logic programming languages operate at a higher level of abstraction: the programmer only needs to be concerned with a logical description of the outcome of her program. Examples of the applications of logic programming languages are:

- Prolog has been used in the JVM 11 specification provide a formal description of semantics of the JVM [13].
- The IBM Watson question-answering system uses Prolog for natural language processing [12].
- Datalog is a subset of Prolog that is used as a database query language. This language has applications in declarative networking, program analysis, security, and cloud computing [7]. For example, the Doop framework uses Datalog for points-to analysis of Java programs [18].

Logic programming languages also typically contain implicit parallelism. For example, Prolog contains *and*-parallelism within each logical statement and *or*-parallelism between logical statements. With the rise of multicore around 2005, it has become increasingly important that programming languages support parallelism. Logic programming languages can be parallelised without needing any explicit primitives for parallelism, so they are well-suited to exploit multicore.

1.2 Project summary

I have implemented a Prolog system inspired by the Warren Abstract Machine (WAM), a popular target for Prolog interpreters. This includes a lexer, parser, and translator to convert a Prolog program to WAM-style instructions, and an abstract machine to interpret these instructions. Pholog diverges from the WAM by adding additional instructions for *cut* and *arithmetic*—key features of any practical Prolog implementation—and does not implement optimisations such as *indexing*. I have also included a static type system, whereas Prolog implementations typically use dynamic type checking.

1.3 Previous work

The first logic programming language was called Baroque, and was implemented in 1972. This was an assembly-like language that used logical statements in the form of Horn clauses [10]. The first Prolog system was also implemented in 1972, by Alain Colmerauer and Philippe Roussel. They chose the name Prolog as an abbreviation for 'programmation en logique' (meaning programming in logic). Robert Kowalski and Maarten van Emden then formalised the interpretation of predicate logic as a programming language [8, 19].

This early work on Prolog was followed by David Warren's invention of the Warren Abstract Machine [20]. The WAM is an abstract machine for efficient execution of a Prolog instruction set: the WAM is analogous to the JVM, but designed for Prolog instead of for Java. The WAM is used by popular Prolog execution systems such as SWI-Prolog [21], and is also used as an intermediate stage when compiling Prolog to native machine code. This project uses a modified version of the WAM instruction set.

A polymorphic type system for Prolog was created by Mycroft and OKeefe [15], based on the Hindley-Milner type system. This type system relies on explicit type annotations for every predicate, but infers the types of variables within predicates.

Chapter 2

Preparation

This chapter contains a discussion of my preparatory work. It begins by discussing Prolog's features, execution model, and program structure (§2.1). The Warren Abstract Machine—a common target for Prolog compilers—is discussed (§2.3), followed by an introduction to the Mycroft-O'Keefe type system (§2.4). After the introduction of these concepts, preparatory work relating to software engineering practice is explained (§2.9), including testing strategy and development methodology.

2.1 Programming in Prolog

A Prolog program is a set of rules, and each rule is a Horn clause: a disjunction of literals (atomic formulae) with at most one positive literal. A left-to-right, depth-first search is performed over these rules to try to find a substitution that makes a given query derivable. Figure 2.1 shows how a Prolog clause relates to a disjunction of literals.

2.1.1 Structure of a Prolog program

A Prolog program is comprised of a set of *predicates*, followed by a *query*. Each predicate is made up of at least one *clause*, and each clause has a *head* containing a list of argument *terms* and a *body* containing a list of *goals*. These concepts are described in more detail below.

Predicate: Predicates are the 'functions' of logic programming languages. A predicate is a set of clauses with a particular name and arity. A goal corresponding to a predicate is satisfied by deriving any one of the clauses in the predicate.

Clause: Each clause represents a logical statement that is true in the world described by the program. A clause consists of a head and a (possibly empty) body. The head is a predicate name with a list of argument terms, and the body is a list of goals. The meaning of a clause is 'if all the body goals can be derived, then the head can be derived'.

Query: A query is a list of initial goals to be derived by the execution of a Prolog program. The execution of a program is a search for a substitution to the query's variables that gives a satisfying interpretation for all the goals in the query; the result of the execution is either such an interpretation, failure, or an infinite loop.

Goal: A goal consists of a predicate name, followed by a list of argument terms. To execute a goal, Prolog searches for an interpretation of the argument terms (a binding to the variables in the argument terms) that satisfies the predicate.

Term: Prolog's only data structure is the term. A term is either an unbound variable, or a compound term consisting of a name and a (possibly empty) list of argument terms. Compound terms are also known as *structures*.

```
A :- B, C.
```

(A) *Prolog-style prepositional logic statement*

$B \wedge C \rightarrow A$

(B) *Logical statement equivalent to* (A)

$\neg B \vee \neg C \vee A$

(C) *Horn clause equivalent to* (A) *and* (B)

FIGURE 2.1: *Example showing the equivalence between Prolog clauses and Horn clauses.*

2.2 Execution model for Prolog

The logical model for Prolog execution is repeated application of a *resolution* rule, where *unification* is performed between a goal and a clause head for each application of resolution.

2.2.1 Unification

Unification is the operation of finding a substitution that gives the same resultant term when applied to a pair of possibly-distinct terms (A, B). A substitution is a mapping from the variables in a term to terms that replace these variables.

When a clause `clause1(t1,...,tn) :- body.` is called with arguments `a1,...,an`, each `ti` is unified with `ai` before the body calls are performed. Here, 'calling' a clause corresponds to performing resolution with the clause. The individual Prolog terms unify as follows:

- The unification of a variable with any other term succeeds, and binds the value of the variable to the value of the other term. Prolog implementations typically perform unification without the *occurs check*, meaning that a variable will unify with a compound term containing itself, creating a self-referential term.
- Compound terms unify if their top function symbol and arity are the same, and their lists of arguments unify recursively. These lists of argument terms might be empty, in which case the terms unify trivially.

2.2.2 Linear resolution

Resolution is an inference rule that combines two complementary literals. A general statement of resolution with unification for first-order logic is

$$\frac{\neg H \vee B_1 \vee \cdots \vee B_n \qquad F \vee G_1 \vee \cdots \vee G_m}{(B_1 \vee \cdots \vee B_n \vee G_1 \vee \cdots \vee G_m)\sigma} \quad \text{if } F\sigma = H\sigma \tag{2.1}$$

for positive literals F and H, other (positive or negative) literals G_i and B_i, and some substitution σ that unifies F and H.

Linear resolution is a technique for first-order logic theorem proving where resolution is repeatedly applied, with each resolvent being used as a parent clause for the next resolution step. The basic execution mechanism of Prolog uses SLD (Simple Linear Definite clause) resolution; a definite clause is a Horn clause with at most one positive literal. This is a special form of linear resolution for Horn clauses based on Kuehner and Kowalski's SL-resolution [9]: this performs linear resolution using a *selection function* to choose the literal to be resolved. In the case of SLD resolution, one of the literals to be resolved is the head of a clause and the other is a goal. A restricted case of Rule 2.1 used by Prolog is

$$\frac{\overbrace{H \texttt{:-} B_1, ..., B_n}^{\text{clause}} \qquad \overbrace{G_1, ..., G_m}^{\text{old goals}}}{\underbrace{(B_1, ..., B_n, G_2, ..., G_m)\sigma}_{\text{new goals}}} \quad \sigma = \text{unify}(H, G)$$

This is equivalent to Rule 2.1 with the constraint that G_i and B_i are negative literals. This constraint arises because a Prolog goal cannot be negated (in pure Prolog).

2.2.3 Search strategy

Prolog performs a depth-first, left-to-right search for the solution to a query: when trying to satisfy a list of goals, the first goal must be satisfied before Prolog progresses to the second goal. It is important that a logic programming language specifies its search strategy because the same program may give a different ordering of results when it is executed with different search strategies, and may have different termination properties. There are advantages to other search strategies over depth first search: breadth-first search will always find a solution if one exists, whereas depth-first search may get stuck in a loop (due to a branch in the search tree that does not terminate).

2.2.4 Backtracking

When Prolog cannot satisfy a goal, *backtracking* occurs. The Prolog system reverts to the most recent resolution step where there was a choice for the clause to resolve against, and selects a different (still unexplored) clause. Resolution steps where another choice of clause remains to be taken are known as *choice points*. A Prolog clause fails to be derived only when there are no remaining choice points to try.

2.2.5 Cut

Cut, written `!`, is a predicate that may appear in the body of a clause. The effect of cut is to:

1. commit to the current clause, meaning that no other clause from the same predicate can be backtracked to within this predicate call,
2. and discard all choice points created by executing goals in the current clause that occur before the cut.

Cut is known as an extra-logical predicate because it does not have a declarative reading. Cut is, however, useful for improving the efficiency of Prolog programs because it can prevent unnecessary backtracking. Cut also increases the expressivity of Prolog, for example cut can be used to implement logical negation.

2.2.6 Negation

Pure Prolog—where each clause is a Horn clause—does not allow the negation of a goal. However, cut can be used to implement a form of negation. For example:

```
unify(X,X).
not_unify(A,B) :- unify(A,B), !, fail.
not_unify(A,B).
```

This demonstrates that explicit negation is not an essential feature for a Prolog system, so I did not implement it.

2.3 The Warren Abstract Machine

The Warren Abstract Machine (WAM) was designed by David Warren in 1983 [20] and has become the standard target for Prolog compilers. It defines an instruction set that can be efficiently interpreted and serves as a useful intermediate representation when compiling Prolog to native code.

This project compiles to a WAM-based, bytecode-style instruction set and interprets these instructions. WAM-like code is the most popular and well-documented target for logic programming languages, making it a good choice for this project. I chose not to target machine code, principally because I expected an interpreter to be faster to implement, giving more

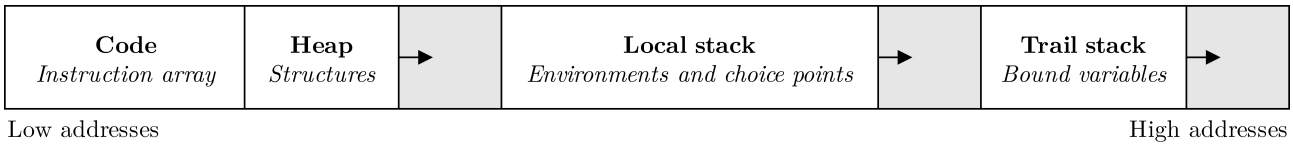


FIGURE 2.2: *Memory layout of the WAM.*

time to spend on other features. Other advantages of byte-code compilation include increased portability and ease of debugging.

2.3.1 Variable representation

An important feature of the WAM is that it represents variables using pointers: a variable is unified with another value by updating the variable to point to the other value. Multiple applications of unification can create chains of pointers.

2.3.2 Instructions

The WAM instruction set contains five main types of instruction:

1. *Get* instructions, used to fetch the values of a clause's formal parameters.
2. *Put* instructions, used to set the actual parameters for a predicate call.
3. *Unify* instructions, used to perform unification on a structure's arguments.
4. *Procedural* instructions, used for control transfer and environment allocation.
5. *Indexing* instructions, used to link the clauses defining a predicate.

These instructions are interpreted by the WAM.

2.3.3 Memory areas

The WAM has three main areas of memory:

1. The *local stack* contains environments and choice points. Environments contain the state associated with the call to a clause: they are similar to stack frames on traditional architectures. Choice points contain the information needed for backtracking.
2. The *heap* contains structures and lists; these are pointed to by variables in the stack.
3. The *trail stack* contains references to variables that were bound on unification. These references are stored so that they can be unbound on backtracking.

The layout of these memory areas is shown in Figure 2.2.

2.4 The Mycroft-O'Keefe type system

The Mycroft-O'Keefe type system [15] is a polymorphic type system for Prolog, based on the Hindley-Milner type system. The Mycroft-O'Keefe type system requires the user to supply type declarations for the program's structures, and to explicitly declare the type associated with each predicate. This information is used to infer types for the program variables.

Type declarations consist of a type name—possibly associated with a list of type variables—followed by a list of different constructors for this type: the declarations are analogous to ML `datatype` declarations. Predicate types consist of a tuple specifying the types of the arguments to the predicate, possibly using type variables. Figure 2.1 compares a polymorphic

```
datatype 'a list = Cons of 'a * 'a list | Nil

fun append(x,Nil) = Cons(x,Nil)
  | append(x,Cons(y,z)) = Cons(y,append(x,z))
```

(A) *Append program in ML*

```
type list A = cons(A,list(A)), nil.

pred append(B, list(B), list(B)).

append(X,nil,cons(X,nil)).
append(X,cons(Y,Z),cons(Y,Res)) :- append(X,Z,Res).
```

(B) *Append program in Prolog using the Mycroft-O'Keefe type system*

LISTING 2.1: *Comparison between an append function implemented in ML and using the Mycroft-O'Keefe type system.*

implementation of append in ML to a polymorphic Prolog implementation using the Mycroft-O'Keefe type system. Before implementing the Mycroft-O'Keefe type system, it was important to understand the exact semantics of this type system. Formal rules are presented in Figure 2.4, along with brief explanations.

Whilst Prolog implementations do not typically use this type system, there are some existing implementations that do use it. For example, the Ciao language is a variant of Prolog that includes an optional type system [6].

2.5 Choice of tools

2.5.1 OCaml

This project aimed to implement Prolog in a higher-level language with features such as garbage collection. I chose OCaml in particular, because it has:

Garbage collection: In general, having a garbage collector simplifies memory management. Garbage collection is particularly useful for this project because the abstract machine implementation can use OCaml's garbage collector to manage its run-time stacks.

Lexer and parser generator tools: OCaml has good implementations of the standard lexer and parser generator tools `lex` and `yacc`, called `Ocamllex` [1] and `Menhir` [14]. It was simpler to use these tools than to implement a lexer and parser from scratch, particularly because the grammar was modified throughout the development of the project.

Library support: There is good library support for OCaml, both from its standard library library and from external libraries such as the Jane Street core library. For this project, the Jane Street core library was used extensively to provide performant implementations of standard datatypes.

Powerful type system: OCaml's type system facilitates rapid development thanks to features such as static type checking and parametric polymorphism. Parametric polymorphism means functions work across multiple datatypes without any extra effort, and static type checking allows many program errors to be detected at compile time.

Support for modules: OCaml includes support for modules and functors. These features enable the interpreter implementation to avoid using a set of mutually recursive functions

$$\Gamma \vdash X : \tau \quad \text{if } X : \tau \in \Gamma \tag{2.2}$$

$$\frac{\Gamma \vdash t_1 : \theta(\tau_1) \ldots \Gamma \vdash t_k : \theta(\tau_k)}{\Gamma \vdash f(t_1, \ldots, t_k) : \theta(\tau')} \quad \text{if } f : \tau_1 \times \tau_k \to \tau' \tag{2.3}$$

$$\frac{\Gamma \vdash t_1 : \theta(\tau_1) \ldots \Gamma \vdash t_k : \theta(\tau_k)}{\Gamma \vdash p(t_1, \ldots, t_k) \text{ Atom}} \quad \text{if } p : Pred(\tau_1, \ldots, \tau_n) \tag{2.4}$$

$$\Gamma \vdash \epsilon \text{ Formula} \tag{2.5}$$

$$\frac{\Gamma \vdash A \text{ Atom}}{\Gamma \vdash A \text{ Formula}} \tag{2.6}$$

$$\frac{\Gamma \vdash \phi_1 \text{ Formula} \quad \Gamma \vdash \phi_2 \text{ Formula}}{\Gamma \vdash \phi_1, \phi_1 \text{ Formula}} \tag{2.7}$$

$$\frac{\Gamma \vdash t_1 : \theta(\pi_1) \ldots \Gamma \vdash t_k : \theta(\pi_k) \quad \Gamma \vdash \phi \text{ Formula}}{\vdash p(t_1, \ldots, t_k) :- \phi \text{ Clause}} \quad \text{if } \Gamma \text{ specifies exactly one type (possibly a type variable) for each variable in the clause, } \theta \text{ is a } \textit{renaming} \text{ substitution, and } p : Pred(\pi_1 \times \cdots \times \pi_k) \tag{2.8}$$

$$\frac{\vdash C_1 \text{ Clause} \quad \cdots \vdash C_n \text{ Clause}}{\vdash C_1 \ldots C_n \text{ Program}} \tag{2.9}$$

(A) *Type rules for the Mycroft-O'Keefe type system [11]*

Type rules	Explanation
(2.9)	A program consisting of a list of well-typed clauses is itself well-typed.
(2.8)	A clause is well-typed if there exists an assignment of types to variables Γ such that: the clause body is well-typed under Γ; and the clause head has a type *equivalent* to the declared type for the clause, up to renaming.
(2.5 – 7)	The body of a clause is well-typed if it consists of a list of well-typed atoms.
(2.4)	A predicate call is well-typed if the arguments have types equal to the predicate's type, with some substitution applied: the arguments types must be some consistent sub-type of the predicate type.
(2.3)	Structure types are some consistent instantiation of the structure's type declaration.
(2.2)	The context Γ specifies types for variables.

(B) *Explanation of type rules*

FIGURE 2.4: *Formal specification of the Mycroft-O'Keefe type system.*

Dependency	Why it was chosen
OCamllex **Menhir**	The lexer and parser generators `Ocamllex` and `Menhir` were used. An alternative to `Menhir` is `Ocamllyac` [1]. `Menhir` was chosen because it gives more comprehensible errors [14]. Using automated tools to generate the lexer and parser enabled me to spend time on other features of the project, and to quickly change the Prolog grammar when adding new features.
OUnit	The OCaml unit test framework `OUnit` [2] was used, avoiding me having to re-implement features of the test framework.
Core	Jane Street's Core OCaml library [3] provided performant implementations of standard data structures such as hash tables. Using this library avoided me having to re-implement these basics.

TABLE 2.1: *Justification of project dependencies.*

in a single file (§3.6).

Other dependencies of the project are justified in Table 2.1.

2.6 Licensing

Licences for the external dependencies of the project are:

- Jane Street Core library: MIT License
- OCaml (including ocamllex): GNU Lesser General Public License, version 2.1
- menhir: GNU General Public Licence
- OUnit: MIT License

These licences all permit the respective dependencies to be used in my project.

2.7 Requirements analysis

The requirements for the core deliverable are stated in the proposal (Appendix C). In summary, the project core consists of a correct implementation of: a lexer and parser; a translator from the parse tree to a byte-code style instruction set; and an abstract machine to execute these instructions. These components are non-negotiable: the project would be considered a failure without them, and tests must be written to demonstrate that the components work.

There are also additional 'nice-to-have' features for the project. In my proposal (Appendix C), I suggested last-call optimisation, determinacy analysis, mode analysis, and type checking as possible additional features. I approached the project with the goal of making a genuinely useful Prolog system, leading me to identify additional nice-to-have features of cut and arithmetic. I did not implement all 'nice-to-have' features, but a justification of those I did implement is:

- **Cut and arithmetic**

  All practical implementations of Prolog include cut and arithmetic, even though the core WAM design does not, so I included them in Pholog (§3.3.2, §3.3.3).
- **Last-call optimisation**

Last-call optimisation is a generalisation of tail recursion optimisation to logic programming languages. The optimisation is relatively simple to implement (§3.3.1), and can provide significant performance benefits (§4.5), so all serious Prolog implementations include this optimisation. I therefore added it to my implementation.

– **Type checker**

The addition of a type system has potential to improve the efficiency of Prolog programmers by providing static error detection. A type system was implemented (§3.5), and the type system is evaluated in terms of correctness and utility (§4.6).

I did not implement determinacy analysis because the analysis scheme I had planned to follow would not have given performance improvements, due to interactions with cut. I also did not implement mode analysis. This was only due to time constraints; mode analysis would have been a useful addition to Pholog.

2.8 Starting point

The full starting point for my project is given in the proposal (Appendix C). A summary is:

– I had studied the Prolog course independently, as it had not yet been taught in Tripos

– I had not implemented a Prolog interpreter or compiler before (or any large Prolog project)

– I had not used OCaml before, but had used ML

2.9 Development methodology

The project was implemented using an *iterative* development methodology, characterised by cycles of development to implement small components of a large system. Each development cycle consists of feature design, development, and testing. Using iterative development methodology enabled me to use all of the available time to improve my project, without risking an incomplete project.

My first development cycle was implementing basic versions of all core components as needed to meet the success criteria, to minimise the risk of not completing these crucial components. In total, I used five development cycles:

1. Core deliverable: lexer, parser, translator, and abstract machine
2. Adding arithmetic
3. Adding cut
4. Last-call optimisation
5. Type-checking

Figure 2.2 shows when these iterations happened, compared with the planned project schedule.

2.9.1 Testing strategy

My development methodology included writing per-component tests and system-wide tests continuously. Each project component had its own test directory, containing component-specific tests. These tests were useful because they enabled bugs to be quickly located, and because the components' output could be directly inspected. A 86 system-wide tests were also written, to ensure that the components correctly implemented Prolog when combined. A test was written to correspond to every bug encountered during development; these tests ensured that the bugs would not reoccur in the final implementation.

13th Nov	**Initial versions of the lexer and parser implemented** The use of `ocamllex` and `menhir` enabled me to implement a lexer and parser quickly, meeting my first major milestone.
18th Nov	*Target for implementation of lexer and parser*
10th Dev	**Translator implemented** I produced an initial translator in time for the milestone, although bugs in this translation phase were uncovered once the abstract machine was working.
16th Dec	*Target for implementation of translator and abstract machine*
20th Dec	**Abstract machine implemented (core deliverable complete)** I met my core deliverable slightly behind schedule, but well before the end of my slack time. The inclusion of slack time prevented the delay from cascading through the rest of the project.
27th Dec	**Arithmetic added** I added support for arithmetic, using planned extension time. This was not part of the core deliverable or a planned extension, but I realised that it is an important feature of any practical Prolog implementation. With hindsight, I should have explicitly scheduled arithmetic as an extension in the proposal.
1st Jan	**Cut added** Cut is another feature not explicitly planned in the project proposal; I added this feature because it is important for writing performant programs, and increases the expressive power of the language. With hindsight, it would have been better to schedule time for adding cut as an extension in the proposal.
19st Jan	**Last-call optimisation implemented** Last-call optimisation was a planned extension in my proposal, which I added during the planned time for extension features.
20th Jan	*End of slack time for core deliverable*
24st Feb	*Target for first dissertation draft*
7st Mar	**Type system implemented** My initial plan did not include adding extension features throughout February, but I continued to implement a type system in parallel with writing a dissertation draft and the progress report.
15th Mar	**First dissertation draft completed** I completed my first dissertation draft about three weeks behind schedule, due to spending unplanned time adding a type system during Lent term. There was, however, enough time remaining before the submission deadline for several iterations of drafting. If I planned the project again, I would move the dissertation draft completion target closer to this date: I think that the work over Lent term has significantly improved my project.
17th May	*Dissertation deadline*

Core component (on time) Core component (behind schedule) Extension component

TABLE 2.2: *Project implementation timeline.*

2.10 Summary

- Prolog terms may be *variables* or *structures* (§2.1.1). A *structure* has a name and a list of arguments.
- A Prolog program consists of a set of clauses followed by a query (§2.1.1). A program is executed by performing a depth-first search over the clauses for a binding to the variables in the query such that the query can be derived.
- The fundamental operation of Prolog is unification (§2.2.1). The unification of a pair of terms applies a substitution to the terms so that they become equal.
- The Warren Abstract Machine (WAM) is a common target for Prolog compilers (§2.3). It defines an instruction set for Prolog that can be efficiently interpreted.
- Prolog can be typed using the Mycroft-O'Keefe type system (§2.4). This is a polymorphic type system similar to that of ML.
- An iterative development methodology was used to gradually add features (§2.9). Tests were written for each system component, and for the system as a whole.

Chapter 3

Implementation

This chapter starts with a discussion of the lexer and parser (§3.1). Pholog's abstract machine architecture is then described (§3.2), and is followed by a description of its instruction set (§3.3). I then describe how the code generation process converts a Prolog program into a sequence of instructions. This is followed by a description of the type-checker (§3.5) including a specification of the formal rules of the type system, and a description of the type-checking algorithm. A repository overview (§3.6) explains the implementation's structure and the testing strategy used. The Pholog pipeline is given in Figure 3.1.

3.1 Lexer and parser

Pholog's lexer and parser were implemented using `ocamllex` [1] and `menhir` [14]. These tools enabled rapid changes to the grammar, supporting my iterative development methodology. The tools automatically generate a lexer and a parser from a specification: the lexer is specified by regular expressions for each of its lexemes, and the parser is specified by a set of context-free grammar rules augmented with information on operator precedence and associativity.

3.2 Abstract machine architecture

I begin by discussing the architecture used by the Pholog interpreter. This section is followed by a discussion of the instructions executed using this architecture. The architecture presented here is similar to the WAM, but differs in its memory layout for *environments* and *choice points* in order to simplify environment deallocation (§3.4.5).

3.2.1 Term representation

Terms are either *variables* or *structures*. *Variables* are represented by references to either a tag representing an unbound variable, or to the value of a variable. Upon unification with another term, the variable updated to point to the term and the variable is written onto the *trail stack* so that the binding can be undone on backtracking. *Structures* are represented by a unique structure ID, followed by an argument array. This term representation is an adaptation of the representation used by the WAM. The type used to represent terms is:

```
type heapValue = InitalValue
               | UnboundVariable
               | Struct of int * heapValue array
               | Int of int
               | BoundTo of heapValue ref
```

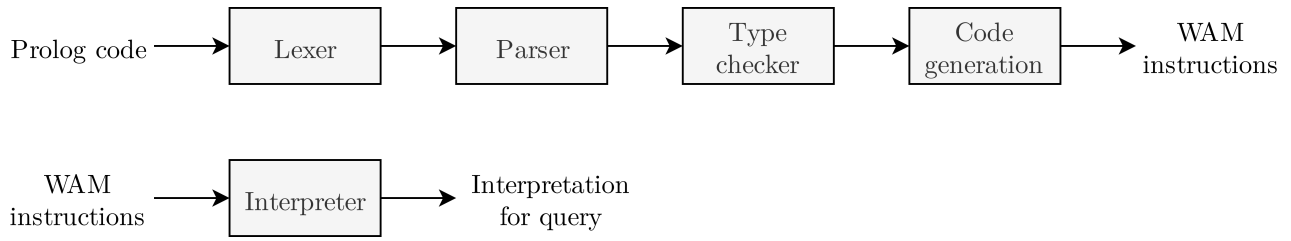
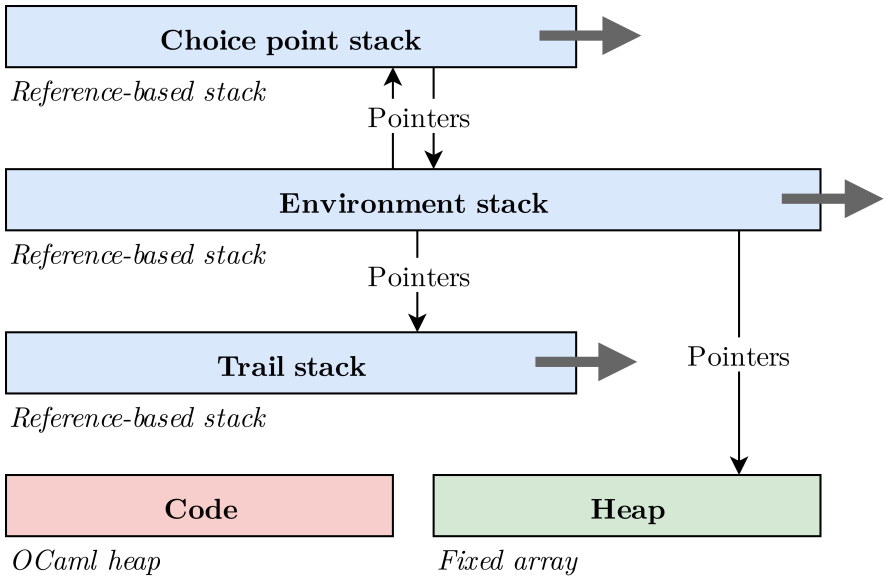
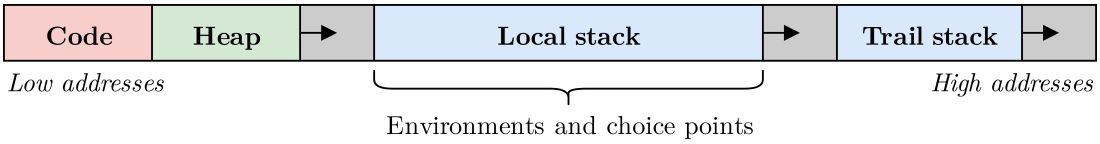


Figure 3.1: *High-level Pholog pipeline.*



(A) *My interpreter*



(B) *The WAM*

FIGURE 3.2: *Comparison between the memory layout used my interpreter and by the WAM.*

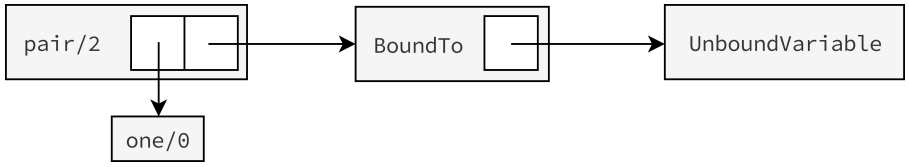


FIGURE 3.3: *Representation of the term* `pair(one,Y)`.

This type includes `InitialValue`: this is used when a structure's arguments are being initialised, and is always overwritten with another value. `InitialValue` is used so that implementation bugs from not initialising part of a structure are easy to detect. This is a addition to the WAM term representation, arising from the switch to OCaml. Figure 3.3 shows the memory representation for the term `pair(one,Y)` where `Y` is unbound; this term is a structure containing a variable.

Variable aliasing

Figure 3.4 shows the unification of two unbound variables. This unification introduces the constraint that the variables must be bound to the same term, but does not specify this term: the variables become *aliased*. When an unbound variable `A` is unified with another unbound variable `B`, the pointer chain associated with `A` is followed up to a `BoundTo` node that points to an `UnboundVariable`. The pointer chain associated with `B` is also followed to reach a `BoundTo` node that points to an `UnboundVariable`. One of these nodes is then updated to point to the other.

On the binding of two unbound variables, it is not necessary to follow *both* variables to reach an `UnboundVariable`: an alternative, correct implementation would be to follow one pointer chain

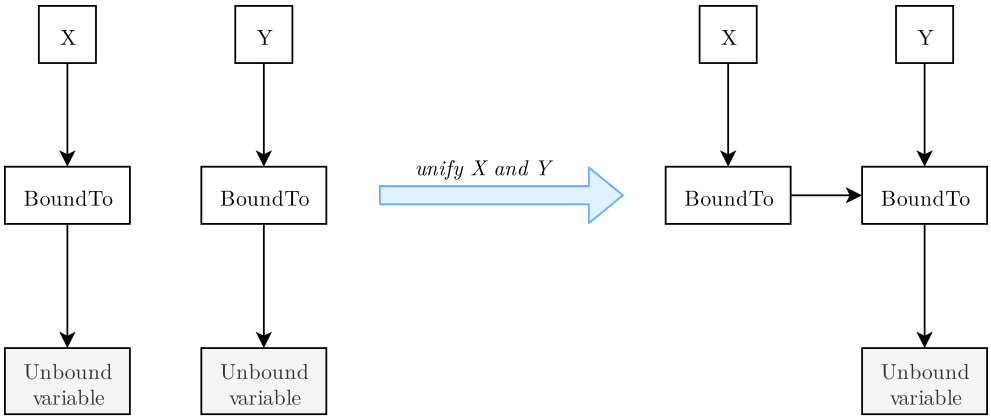


FIGURE 3.4: *The aliasing of variables on unification.*

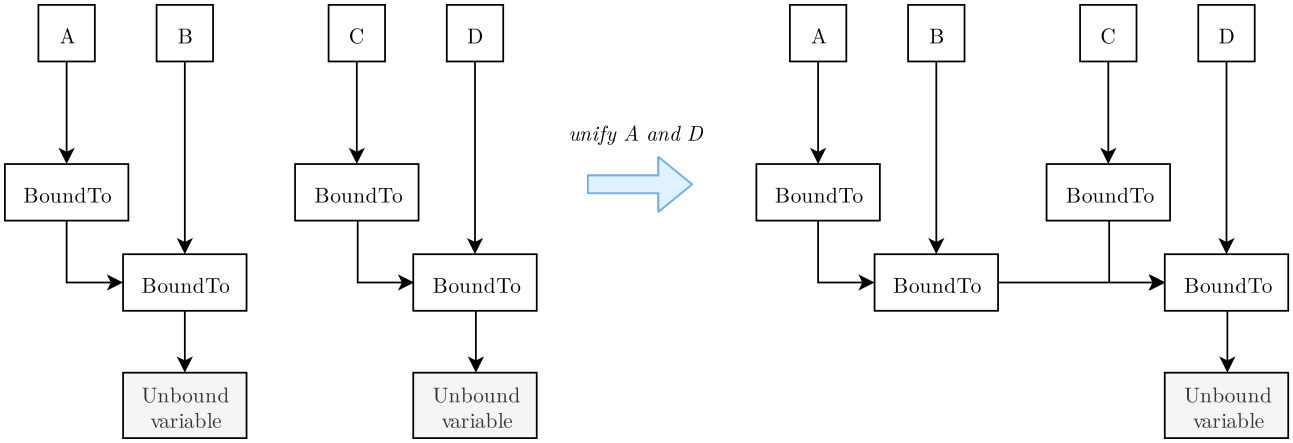


FIGURE 3.5: *Creation of a pointer chain of length three.*

---

to an `UnboundVariable` and bind this to the head of the other pointer chain. This approach was not used because it gives longer pointer chains—slowing subsequent unifications and look-ups—but it would have the advantage of decreasing the time taken by the first unification. Successive unifications can create arbitrarily long pointer chains, and these chains cannot be collapsed without knowing which nodes are pointed to. A diagram showing how long pointer chains are created is given in Figure 3.5.

3.2.2 Memory layout of the Pholog interpreter

Figure 3.2 summarises the memory layout for the interpreter, and gives a comparison to the WAM. This section discusses the memory areas shown in Figure 3.2 in more detail.

Environments

The *environment stack* contains the *environments* of the computation. There is an environment associated with each clause evaluation; in this sense, the environment stack is analogous to the call stack of a conventional architecture. An environment contains state associated with the evaluation of a clause, including:

- The values of the non-temporary variables in the clause.
- The clause's return address; this is written to the program counter when the clause returns.
- Pointers to the heads of the *choice point* and *trail* stacks, needed for cut (§3.2.3).
- A pointer to the environment of the clause's parent, needed when the clause returns.

The environment 'stack' is not truly a stack; it is more accurately described as a directed acyclic graph, because pointers from the choice-point stack keep multiple 'heads' of the environment stack live. These heads enable reverting the environment stack to a previous state on backtracking. Figure 3.6 explains an example program where an environment needed for backtracking cannot be garbage collected.

This is a divergence from the WAM, which uses a stack (implemented as a continuous region of memory) containing both environments and choice points. This is called the local stack, and is shown in Figure 3.2. I did not use the WAM-style local stack because Pholog is implemented in a garbage-collected language rather than a C-like language: environment deallocation is simplified by using Pholog's stack representation in combination with garbage collection (§3.4.5).

The heap

A program's terms are stored in OCaml's heap, and are not grouped into any conceptual region.

Choice points

Choice points are used to store information used for backtracking. There is a choice point for each predicate with more than one clause; a choice point is not needed for a predicate consisting of a single clause. Each choice point contains information needed to restore the computation to its state when a predicate was called, so that a different choice can be made for which clause to invoke. This state includes:

- A pointer to the environment that called the predicate, needed in order to restore the environment stack to its previous state (just before the predicate was called). The environment stack is represented by a chain of pointers, so a pointer to a single environment is enough to restore the whole environment stack.
- The arguments to the predicate. These must be restored to the argument array on backtracking, because they might have been overwritten by the time backtracking occurs.
- The address of the first instruction of the next clause from the predicate to try: this is written to the program counter on backtracking.
- The instruction address to return to once the predicate finishes execution.

The choice-point stack has a similar representation to the environment stack: each choice-point node has a pointer to its predecessor. This enables restoration of the choice-point stack to an earlier state by updating the stack head to a previous value (this is needed to implement cut). The choice-point stack was represented by a simple OCaml list before cut was implemented.

The trail

The trail stack contains a record of variables that have been bound during unification; these are unbound on backtracking. The trail stack also contains markers that determine which variables on the trail stack to unbind: when backtracking, variable bindings on the trail are reset up to the first marker.

It is natural to use the same data structure for the trail stack as for the choice-point stack, because backtracking to a choice point requires knowing which bindings on the trail stack need to be undone: a pointer into the choice-point stack is not useful without a corresponding pointer into the trail stack. The trail stack is, therefore, also implemented by having each stack element point to its predecessor.

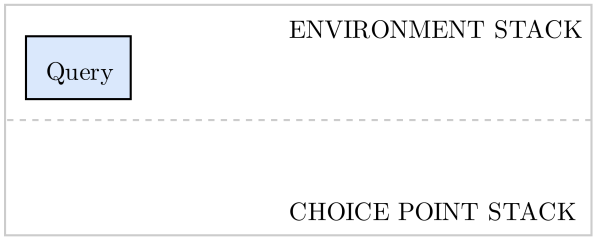
Computation state

The Pholog interpreter must track various values at run-time. This includes maintaining space for arguments and temporary variables, as well as passing information between environments (such as the return address).

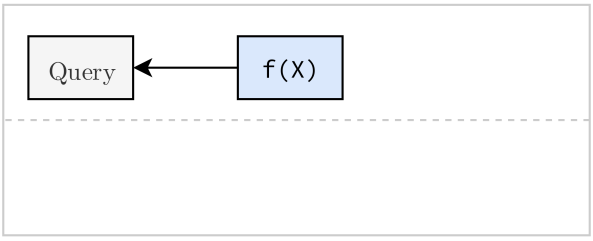
```
g(1).
g(2).
f(X) :- g(X).

?- f(X), X is 2
```

(A) *A Prolog program that backtracks*

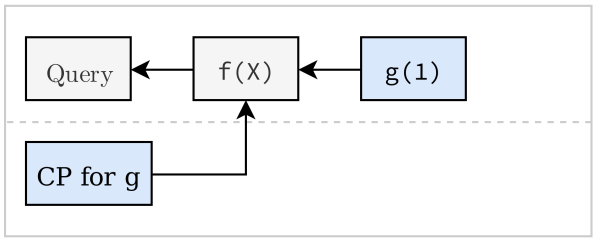
1. The environment for the query is the initial head of the environment stack.



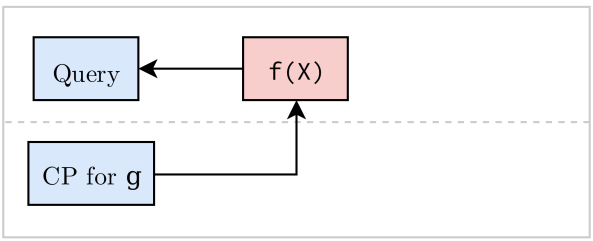
2. `f(X)` is called with `X` an unbound variable.



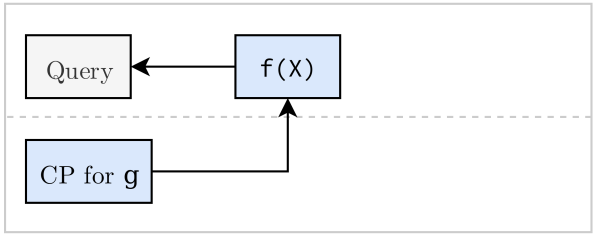
3. `f(X)` calls `g(1)`, the first clause in `g`.



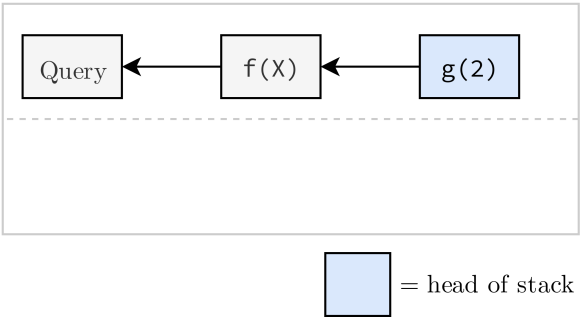
4. `g` and `f` return, so the query is now the head of the environment stack, but the environment for `f(X)` is not deallocated.



5. `X is 2` fails, causing backtracking to `g`.



6. `g(2)` is called upon backtracking, all clauses will now return successfully.



(B) *A sequence of steps in the evaluation of the program in* (A)

FIGURE 3.6: *Diagram showing how pointers into the environment stack are used to implement backtracking. At step 4, the head of the environment stack is the environment for the query, but the environment for* `f(X)` *is not garbage-collected because it is pointed to from the choice-point stack.*

An array is reserved as space to store temporary variables, with size given by the largest number of temporary variables in any clause. Similarly, an array is reserved to store the arguments to a predicate call, allowing the arguments to be accessed at a fixed and known location. A struct is maintained to keep track of other aspects of the computation state, including:

- The current value of the program counter.
- A pointer to whichever structure on the heap is currently being built, or was most recently being built. I refer to this as the *structure pointer*.
- A return address register, used to populate the return address of the next environment created.

3.2.3 Cut

The interpreter memory layout is important to the implementation of *cut*, an operator used to restrict backtracking. The effect of a cut in the body of the clause is to:

1. commit to the current clause,
2. and commit to all choices made so far in the execution of the current clause.

This is achieved by resetting the heads of the choice-point and trail stacks to where they were before the predicate was called. To achieve this, the environment stack contains pointers into these stacks that can be used to reset the stack heads.

3.2.4 Driver function for the abstract machine

The abstract machine interprets a sequence of instructions using a tail-recursive driver function. This function pattern-matches on the instruction being executed and either performs a tail call to a function to execute the instruction or executes the instruction in-line (for simpler instructions). Each function to execute an instruction will itself perform a tail-call to the driver function: the helper functions and driver function are mutually recursive. However, explicit mutual recursion of functions is avoided using mutually recursive OCaml modules, as discussed in §3.6. An extract from the driver function is given in Appendix B.

The use of tail recursion in OCaml ensures that the abstract machine will never run out of stack space and removes overheads from growing the stack, because the OCaml compiler converts the tail recursion to iteration. To test my implementation, the size of OCaml's run-time stack was measured during the execution of all test programs. It was observed to be constant for any given program, as expected.

3.3 Instruction set

Pholog's instruction set aims to capture the fundamental low-level operations of logic programming languages. This enables the implementation of performant interpreters, as well as providing an intermediate representation when compiling logic programming languages to native machine code. In this section, the 34 different instructions interpreted by Pholog's abstract machine are discussed. The full instruction set is summarised in Table 3.1; the rest of this section discusses key points of the instruction set.

3.3.1 Last-call optimisation

When a clause's final instruction calls another predicate, the clause's environment can be deallocated before the final call. Early deallocation of a final call is known as *last-call optimisation*, and was implemented as an extension to this project. My implementation of the optimisation is evaluated in §4.5.

Instructions	Description
Calling predicates `Call, CallAfterDealloc`	Call a predicate by setting the program counter to the predicate's starting address. `CallAfterDealloc` is used for last-call optimisation (§3.3.1).
Allocating environments `Allocate`	Allocate the environment for a clause, given the environment's size. Update the environment stack head to the new environment.
Deallocating environments `Deallocate, DeallocateBeforeLastCall`	Reset the environment stack head to the clause's predecessor. `DeallocateBeforeLastCall` is used for last-call optimisation.
Choice point manipulation `TryMeElse, RetryMeElse, TrustMe, RmCps`	Choice points are created by a `TryMeElse` instruction before the first clause in the predicate, updated by `RetryMeElse` instructions before the middle clauses, and removed by a `TrustMe` instruction before the final clause. `RmCps` removes choice points for cut (§3.3.2).
Get clause arguments `GetVariable, GetValue, GetStructArg, GetStructTemp, GetInt`	Used to get the arguments to a clause (§3.4.2). Each instruction has two operands: an argument term, and a term to be unified with it.
Put clause arguments `PutStructArg, PutStructTemp, PutIntA, PutVariable, PutValue`	Analogous to 'get clause argument' instructions, but used to set the arguments to the callee (§3.4.2).
Get structure arguments `StructGetVariable, StructGetValue, StructGetInt`	Get a structure's arguments, as part of getting the arguments to a clause (§3.4.2).
Set structure arguments `SetVariable, SetValue, SetInt`	Analogous to the 'get structure arguments' instructions, but instead used to set a structure's arguments. Used to make predicate calls (§3.4.2).
Is control instructions `InitAcc, Is, PutIntT`	Is-expressions are evaluated with an accumulator (§3.3.3). The `InitAcc` and `PutIntT` instructions initialise this accumulator to a variable or an integer, respectively.
Arithmetic instructions `AddVar, AddInt, SubVar, SubInt`	Used to evaluate the body of an is-expression (§3.3.3). Each instruction specifies a variable/constant to be added/subtracted from the accumulator.
Control instructions `Finish, Backtrack`	`Backtrack` is used to implement a `fail` predicate that always causes backtracking, and `Finish` is used to stop execution.

TABLE 3.1: *Summary of the instruction set.*

In Pholog, last-call optimisation may allow an environment to be garbage collected early: the optimisation reduces memory use. Execution times may also improve due to reduced overheads from expanding the heap. I implemented last-call optimisation by adding additional call and deallocate instructions: `DeallocateBeforeLastCall` and `CallAfterDealloc`, following the approach used by the WAM for last-call optimisation. These instructions are discussed at the end of this section.

However, last-call optimisation is not useful for every clause that finishes with a predicate call. A clause's environment cannot be deleted if any predicate called (directly or indirectly) by the clause might be backtracked to, because the clause's environment would be needed on backtracking. Pholog relies on garbage collection to delete environments: the use of garbage collection ensures no environment is deleted whilst it is still needed. If an environment might be needed on backtracking then there exists a pointer chain from the choice-point stack to the environment, so my implementation of last-call optimisation simply needs to move the environment stack head pointer back by one.

Last-call optimisation is applied even when it does not save space, because the optimisation never has a detrimental effect. This removes the need for program analysis to determine when to apply the optimisation, reducing implementation complexity and compile times. The WAM uses a similar approach, also applying last-call optimisation where it does not save stack space, but manually prevents an environment from being deallocated whilst it is still needed (this is known as *environment protection*).

Comparison between `Dealloc` and `DeallocateBeforeLastCall`

Both instructions reset the environment stack to the current clause's predecessor, but the instructions differ in their other effects:

Effect on the program counter
`Dealloc` sets the program counter to the current clause's return address, whereas `DeallocateBeforeLastCall` needs to increment the program counter so the subsequent `CallAfterDealloc` instruction is called.

Effect on the return address register
`Dealloc` does not change the current return address value (use described in §3.2.2), whereas `DeallocateBeforeLastCall` must set the return address to the current clause's return address. This causes the subsequent predicate call to return straight to the current clause's parent.

Comparison between `Call` and `CallAfterDealloc`

Both instructions update the program counter to a specified instruction, but `Call` must also set the return address register to point to the next instruction in the current clause. `CallAfterDealloc` does not need to change the return address register because it has already been initialised by a preceding `DeallocateBeforeLastCall` instruction.

3.3.2 Cut

Another extension feature that I added is *cut*; this is not present in the WAM. Pholog implements cut by resetting the head of the choice-point stack to where it was before the current clause was called (§3.2.3). The old choice-point stack head is obtained from a field of the current environment, and the `RmCps` instruction is used to reset the choice-point stack to this value.

3.3.3 Arithmetic

I extended Pholog by adding support for integer arithmetic: this is also not part of the WAM. The addition of arithmetic was necessary to make the project practically useful, and it enabled

```
fib(1,0).
fib(2,1).
fib(X,Y) :- X2 is X - 2, X1 is X - 1, fib(X2, Ans2), fib(X1,Ans1), Y is Ans2 + Ans1.
```

LISTING 3.1: *Prolog program using arithmetic inside an* `is` *statement.*

```
InitAcc (Temp 2), (T (Temp 0))  /* accu = Y      */
AddI (Temp 2), 2                /* accu += 2     */
AddI (Temp 2), 3                /* accu += 3     */
Is (T (Temp 1)), (Temp 2)       /* unify(X,accu) */
```

(A) *Instructions to evaluate* `X is Y + 2 + 3`

```
PutIntT (Temp 2), 1             /* accu = 1      */
AddI (Temp 2), 2                /* accu += 2     */
Add (Temp 2), (T (Temp 0))      /* accu += Y     */
Is (T (Temp 1)), (Temp 2)       /* unify(X,accu) */
```

(B) *Instructions to evaluate* `X is 1 + 2 + Y`

LISTING 3.2: *Instructions generated for example arithmetic expressions.*

---

a more realistic comparison to other implementations. Arithmetic is performed within an `is` block, consisting of:

1. a variable or an integer,
2. followed by the keyword `is`,
3. followed by an arithmetic expression.

Listing 3.1 gives an example program with integer arithmetic. The `is` blocks are executed by evaluating the arithmetic expression, giving an integer, and then unifying this integer with the value on the left of the `is` block.

All arithmetic operations supported are left-associative and bracketing sub-expressions is not supported. These restrictions enabled me to implement arithmetic quickly, giving more time to spend on other features of Pholog. An arithmetic expression can, therefore, be compiled to a sequence of addition or subtraction instructions performed on a single accumulator. The `InitAcc` and `PutIntT` instructions are used to initialise this accumulator. Once an arithmetic expression has been evaluated, the instruction `Is` is used to unify the result with some variable. Listing 3.2 gives examples of compiling `is` expressions.

3.4 Code generation

Code generation is performed by the translator, using a number of passes:

– First, the clauses for each predicate are extracted and grouped. As the clauses are extracted, explicit variable names from the clauses are replaced with variable locations. This pass determines whether a variable is temporary, or whether it needs to be stored in a clause's environment (see §3.4.1).

– The instruction sequence for each predicate is generated, using abstract addresses to represent other predicates instead of instruction positions.

– The instructions for all of the predicates are combined into a single array.

```
perm(X,c(H,T)) :- take(X,H,R), perm(R,T).
```

(A) *Before variable abstraction*

```
perm(Temp0, c(Temp1, Env0)) :-
    take(Temp0, Temp1, Env1),
    perm(Env1, Env0).
```

(B) *After variable abstraction*

Program variable	Abstract variable
X	Temp0
H	Temp1
T	Env0
R	Env1

(C) *Mappings from program variables to abstract variables*

LISTING 3.3: *A clause before and after the abstraction of its variables.*

– Finally, abstract predicate addresses are replaced with positions in the array.

The rest of this section explains these steps in more detail, and Figure 3.10 summarises this section by giving an example program and explaining its instructions. A larger example is given in Appendix A.

3.4.1 Translating variables to locations

The first step of the translator is to replace the string variables in each clause with memory locations, and to group these abstracted clauses by predicate `id`. The location for a variable depends on whether or not the variable is temporary; a non-temporary variable is a variable whose value persists over a predicate call. A non-temporary variable is therefore either:

– used in two separate predicate calls in the clause body,

– or used in the clause head, and in a clause body call that is not the first body call.

The non-temporary variables in each clause are assigned a unique position in the clause's environment. These variables need to be stored in the environment so that they persist over predicate calls. The temporary variables are assigned a position in the array of temporary variables. An example of this pass of the translator is given in Listing 3.3.

3.4.2 Generating the instructions for each clause

A clause corresponding to a rule does two main things: it gets its arguments from the argument array, possibly extracting some variables used in the clause, and then makes a series of calls to other predicates. Clauses without body goals may get arguments but do not make any predicate calls. The instructions for a clause follow this structure: first they get the clause arguments, and then the clause makes calls to other predicates, loading values into the argument array for each call.

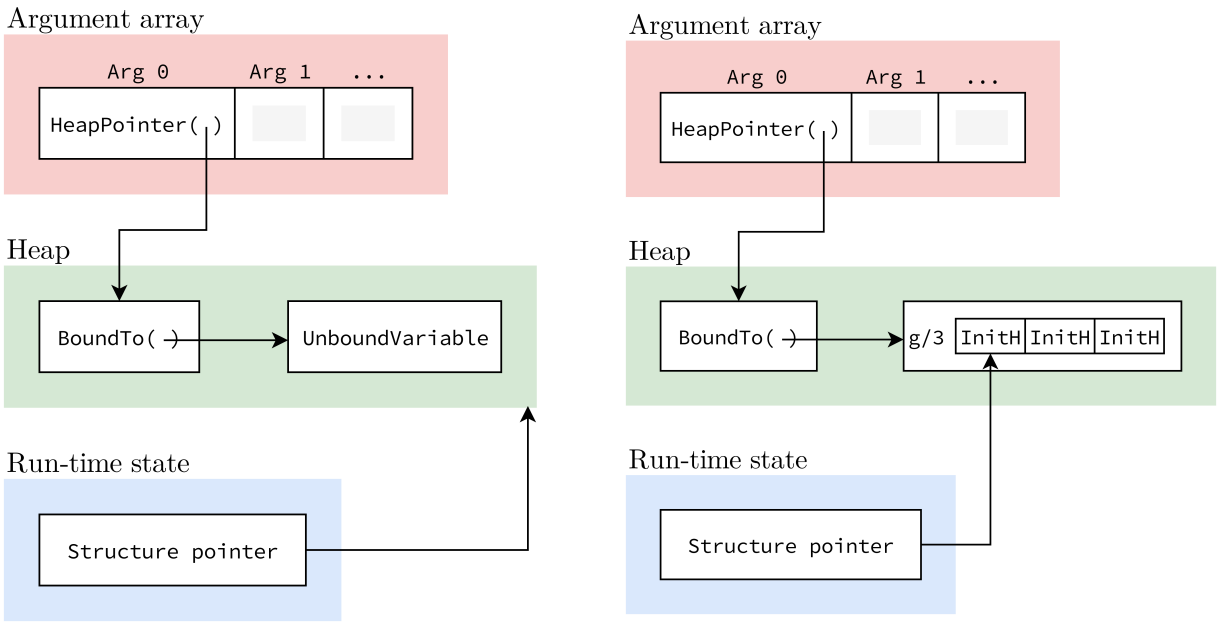
Getting the arguments to a clause

Getting a clause argument that is an integer or variable is simple, and can be achieved with the `GetInt`, `GetVariable` or `GetValue` instruction:

– `GetInt` unifies an argument with an integer literal
– `GetVaraible` unifies an argument with a *new* unbound variable
– `GetValue` unifies an argument with an already-instantiated variable

The complicated case is loading structures, particularly nested structures. The sequence of instructions to load a nested structure is:

1. *Single* `GetStructArg` *instruction*
   The instruction sequence begins with a `GetStructArg` instruction, taking two values: the



(A) *Before* (B) *After*

FIGURE 3.9: *The effect of the instruction* `GetStructArg g/3 <- Arg 0` *when* `Arg 0` *contains an unbound variable.*

```
GetStructArg ((0, 3), (Arg 0))    /* get f/3 from argument 0                    */
StructGetVariable (T (Temp 0))    /* unify temp 0 with X (1st argument of f/3) */
StructGetVariable (T (Temp 1))    /* unify temp 1 with g(1,2)                  */
StructGetInt 3                    /* unify the 3rd argument of f/3 with 3      */
GetStructTemp ((0, 2), (Temp 1))  /* get g/2 from temp 1                       */
StructGetInt 1                    /* unify the 1st argument of g/2 with 1      */
StructGetInt 2                    /* unify the 1st argument of g/2 with 2      */
```

LISTING 3.4: *Instructions to load the arguments of* `cl(f(X,g(1,2),3))`.

`id` of the topmost structure to be loaded, and the argument position to load from. The argument term is unified with the topmost structure, and the structure pointer is set to the first element in the body of the resulting term. Figure 3.9 shows the effect of a `GetStructArg` instruction.

2. *Sequence of* `StructGet` *instructions*
   The `GetStructArg` instruction is followed by a sequence of `StructGet` instructions that each unify a value with the arguments to the topmost structure.

   Nested structures are extracted into temporary variables with `StructGetVariable` instructions, and can then be loaded by repeated application of the instruction sequence explained above. The only change needed is that `GetStructTemp` is used instead of `GetStructArg` when loading a structure from a temporary variable.

An example of an instruction sequence following these rules to load a nested argument structure is given in Listing 3.4.

Making a call to another predicate

Making a call to a predicate requires loading the actual arguments for the call, then performing the call with a `Call` or `CallAfterDealloc` instruction. It is simple to put an integer or variable in an argument position: this can be achieved with a `PutVariable`, `PutValue`, or `PutInt`

```
PutStructTemp ((0, 2), (Temp 1))    /* temp 1 = g/2                    */
SetInt 1                            /* 1st argument of temp 1 is 1     */
SetInt 2                            /* 2nd argument of temp 1 is 2     */
PutStructArg ((1, 3), (Arg 0))      /* arg 1 = f/3                     */
SetVariable (E (Env 0))             /* 1st argument of arg 1 is X      */
SetValue (T (Temp 1))               /* 2nd argument of arg 1 is g(1,2) */
SetInt 3                            /* 3rd argument of arg 1 is 3      */
PutVariable ((E (Env 1)), (Arg 1))  /* arg 2 = Y                       */
PutIntA ((Arg 2), 4)                /* arg 3 = 4                       */
Call (PositionF 2)                  /* call predicate cl               */
```

LISTING 3.5: *Instructions to put values into the argument array for the predicate call* `cl( f(X,g(1,2),3), Y, 4).`

instruction. The more complicated case is loading a structure to a predicate argument.

`PutStructTemp` instructions are used to initialise a structure, and `Set` instructions set the structure's arguments by following the structure pointer. Listing 3.5 gives an example of instructions to call predicate with a nested structure as an argument.

3.4.3 Generating the instruction sequence

Once the instruction sequences for the predicates have been generated, the instructions are combined into a single array. First, an array is initialised with size equal to the total number of instructions. Each clause is copied into this array, and a mapping from clause `id` to array position is maintained. This clause `id` is the 3-tuple of predicate symbol, predicate arity, and clause number within the predicate. Once the instructions are in the array, the instructions are iterated over to replace all abstract addresses with integer positions in the array.

3.4.4 Summary of code generation

The main steps in the code generation process are:

- Each clause is translated to instructions to get the clause's arguments, followed by instructions to call other predicates.
- The clauses in a predicate are linked using control instructions.
- The instructions for all predicates are combined into a single array, and predicates are called using offsets into this array.

Figure 3.10 explains all of the instructions generated for an example program.

3.4.5 Relationship to the Warren Abstract Machine

Pholog's architecture is based on the WAM, and therefore has many similarities. Both architectures use environments and choice points in a stack, represent variables with pointers, and have similar instruction sets. Some key differences are:

- The WAM stores environments and choice points in the same stack, called the local stack, whereas Pholog separates these two stacks: see Figure 3.2. This difference arises because Pholog's abstract machine is written in OCaml, so uses garbage collection for memory management. Separating the stacks and storing pointers between them neatly handles the problem of deallocating environments once they will no longer be needed.

  Safe environment deallocation is more difficult without a garbage collector. To ensure safety, the WAM stores environments and choice points on the same stack: the topmost

```
addOne(X,Y) :- Y is X + 1.
incAll(eol,eol).
incAll(cons(H,T),cons(HNew,TNew)) :- addOne(H,HNew), incAll(T,TNew).

?- incAll(cons(1,cons(2,eol)),Result)
```

(A) *A Prolog program*

```
?- incAll(cons(1,cons(2,eol)),Result)
1.  Allocate 1                          /* Create environment for 1 variable        */
2.  PutStructTemp ((0, 0), (Temp 1))    /* Temp 1 is eol                            */
3.  PutStructTemp ((1, 2), (Temp 0))    /* Temp 0  is cons/2                        */
4.  SetInt 2                            /* 1st argument of Temp 0 is 2              */
5.  SetValue (T (Temp 1))               /* 2nd argument of Temp 0 is Temp 1 (eol)   */
6.  PutStructArg ((1, 2), (Arg 0))      /* Arg 0 is cons/2                          */
7.  SetInt 1                            /* 1st argument of Arg 0 is 1               */
8.  SetValue (T (Temp 0))               /* 2nd argument of Arg 0 is Temp 0          */
9.  PutVariable ((E (Env 0)), (Arg 1))  /* Arg 1 is Result                          */
10. Call (PositionF 12)                 /* Call incAll/2                            */
11. Finish                              /* Finish execution                         */

incAll(eol,eol).
12. TryMeElse (PositionC 17)            /* Try clause at addr 17 on backtracking    */
13. Allocate 0                          /* Allocate environment with no variables   */
14. GetStructArg ((0, 0), (Arg 0))      /* Unify Arg 0 with eol                     */
15. GetStructArg ((0, 0), (Arg 1))      /* Unify Arg 1 with eol                     */
16. Deallocate                          /* Deallocate environment                   */

incAll(cons(H,T),cons(HNew,TNew)) :- addOne(H,HNew), incAll(T,TNew).
17. TrustMe                             /* Remove choice point                      */
18. Allocate 2                          /* Create environment for 2 variables       */
19. GetStructArg ((1, 2), (Arg 0))      /* Unify Arg 0 with cons/2                  */
20. StructGetVariable (T (Temp 0))      /* Bind Temp 0 to 1st arg of Arg 0 (H)      */
21. StructGetVariable (E (Env 0))       /* Bind Env 0 to 2nd arg of Arg 0 (T)       */
22. GetStructArg ((1, 2), (Arg 1))      /* Unify Arg 1 with cons/2                  */
23. StructGetVariable (T (Temp 1))      /* Bind Temp 1 to 1st arg of Arg 1 (HNew)   */
24. StructGetVariable (E (Env 1))       /* Bind Env 1 to 2nd arg of Arg 1 (TNew)    */
25. PutValue ((T (Temp 0)), (Arg 0))    /* Set Arg 0 to Temp 0 (H)                  */
26. PutValue ((T (Temp 1)), (Arg 1))    /* Set Arg 1 to Temp 1 (HNew)               */
27. Call (PositionF 32)                 /* Call addOne                              */
28. PutValue ((E (Env 0)), (Arg 0))     /* Set Arg 0 to Env 0 (T)                   */
29. PutValue ((E (Env 1)), (Arg 1))     /* Set Arg 1 to Env 1 (TNew)                */
30. DeallocateBeforeLastCall            /* Deallocate before final call            */
31. CallAfterDealloc (PositionF 12)     /* Call incAll recursively                  */

addOne(X,Y) :- Y is X + 1.
32. Allocate 0                          /* Allocate environment with no variables   */
33. GetVariable ((T (Temp 0)), (Arg 0)) /* Set Temp 0 to Arg 0 (X)                  */
34. GetVariable ((T (Temp 1)), (Arg 1)) /* Set Temp 1 to Arg 1 (Y)                  */
35. InitAcc ((Temp 2), (T (Temp 0)))    /* Initialise accumulator to Temp 0 (X)     */
36. AddI ((Temp 2), 1)                  /* Add 1 to accumulator                     */
37. Is ((T (Temp 1)), (Temp 2))         /* Unify Temp 1 (Y) with accumulator (X+1)  */
38. Deallocate                          /* Deallocate environment                   */
```

(B) *Instructions generated for the Prolog program*

FIGURE 3.10: *An explanation of the instructions generated for an example program.*

choice point 'guards' all environments beneath it, preventing them from being deallocated. The advantage of Pholog's approach is that its simplicity increases confidence that the implementation is correct, and makes the project easier to maintain and update.

– Pholog includes instructions not found in the WAM to provide additional features: I added instructions used for cut and arithmetic. These additional instructions include `RmCps`, `AddI`, and `Is`. I added these features because they are necessary for practical use-cases, and because they enable a more realistic comparison to other implementations.

– The WAM includes instructions that I have not implemented; these are used for clause indexing. Indexing is an optimisation used to determine which clause from a predicate to execute, based on structure of an argument term. Adding this to Pholog would be likely to improve execution times, but it is not included due to time constraints.

3.5 Type-checking

Type-checking is performed using the Mycroft-O'Keefe type system [15]. Types are manually specified for each predicate, and type declarations (like ML `datatype` declarations) associate types with all structures in a program. Examples are:

Type declaration:	*Predicate type specification:*
`type list A = cons(A,list(A)), nil.`	`pred append(B, list(B), list(B)).`

The types of the program's variables are determined by type inference. Polymorphism is provided by using explicit type variables for predicates and user-defined types. For example, `A` and `B` above are type variables. Formal rules for this type system are given in Figure 2.4 (page 8). The restriction in the typing rule for clauses, given in rule 2.8 of Figure 2.4, is important: the type of a clause head must be equivalent to the user-provided predicate type signature. This restriction arises because a predicate implementation must be as general as its type signature; I refer to this as the *definitional generality* constraint. As an example of this constraint, consider the following predicate:

```
pred unify(A,A).    /* type declaration for unify predicate */
unify(X,X).
```

If the definition of the unify predicate were instead `unify(1,1)` then the program should fail to type-check, since the type declaration for the predicate allows it to be called with arguments that are not integers but the clause definition only applies to integers.

Before any clause is type-checked, the type definitions and predicate signatures are extracted from the source program. The sequence of steps for type-checking a clause is:

1. *Collect all of the types that each variable must satisfy*
   Build a list of the types associated with all of the places a variable is used.

2. *Infer variable types*
   Unify the types associated with each variable to obtain an inferred type for the variable. This is complicated by the definitional generality constraint: see §3.5.1. Type aliasing (when two type variables are unified) is ignored at this stage, since it is handled at the next step. Figure 3.11 gives an example of this inference, and Figure 3.12 gives an example of a structure type being inferred that is never explicitly present in the program.

3. *Type-check clause body calls*
   Check that each predicate call has argument types that are equal to the predicate type declaration with some substitution applied. This substitution need not be renaming: it may instantiate type variables in the predicate type declaration.

```
pred any(A).
any(V).

pred unify(A,A).
unify(V,V).

pred f(int,A).
f(X,Y) :- any(X), any(Y), unify(Y,Z).
```

(A) *A typed Prolog program*

Variable	List of its types	Unification of the types
X	int Unnamed1	int
Y	A Unnamed2 Unnamed3	A
Z	Unnamed4	Autogen4

(B) *The types inferred for the variables in* (A) *during the first pass of the type-checker*

FIGURE 3.11: *An example of the first pass of the type-checker, where variable types are inferred.* `Autogen` *types are used for the unification of unnamed type variables.*

A predicate call may cause the aliasing of two type variables, and this aliasing will not be captured by the earlier type inference step. It is necessary to maintain a binding between variables at this step to record type aliases generated by predicate calls. Figure 3.13 gives an example of an ill-typed clause that would type-check if type variable aliasing was not considered.

4. *Type-check clause head*
   Check that the clause head has the same type as the predicate type declaration. These types only need be equivalent up to renaming to satisfy definitional generality, but my implementation does not rename types from the clause head so renaming does not need to be considered.

3.5.1 Enforcing definitional generality

In order to ensure that a clause's type is as general as its predicate type declaration, I use the concept of *named* and *unnamed* type variables. When type-checking a clause, type variables from the clause's type declaration are the *named* type variables and all others are *unnamed* (see Figure 3.11 for an example).

Unification of type variables in step 2 is subject to the constraint that a named typed variable will only unify with another type variable of the same name, or an unnamed type variable. When a list of types are unified to infer a variable's type, if this list contains a named variable then the result is either the named variable or failure. Similarly, a named type variable will not be bound to any other type variable in step 3. This prevents a clause becoming less general than its declared type due to binding of type variables in the predicate type declaration, as illustrated by Figure 3.13.

3.6 Repository overview

All source code was written from scratch, although the lexer and parser use `ocamllex` and `menhir`. The OCaml source code is grouped into libraries containing the project components, and each library exposes an interface to the other libraries using a `.mli` file. In OCaml, every `.ml` file corresponds to a module, and files may define additional modules in their body. OCaml also supports *functors*: modules that are parametrised on other modules. Figure 3.14 shows the dependencies between the libraries in the project, and also includes the dependencies between modules for the key libraries `typecheck`, `code_generation`, and `execute`.

```
type color =          type f A, B =
red, green, blue.     mkf(A, B).

pred p1(f(int,A)).    pred p2(f(A,color)).
p1(mkf(1,X)).         p2(mkf(X,red)).

?- p1(Y), p2(Y)
```

(A) *A typed Prolog program using a variable* `Y`

Types associated with `Y`	Unification of these types
`f(int, Unnamed1)` `f(Unnamed2,color)`	`f(int,color)`

(B) *Type unification to infer the type of* `Y`*; this type is not present anywhere in the program*

FIGURE 3.12: *An example showing how the first type-checker pass can infer a structure type.*

```
pred unify(A,A).
unify(X,X).      /* This predicate must have two arguments with the same type      */

pred p(A,B).
p(X,Y) :-        /* X has type A, and Y has type B                                 */
  unify(X,Z),    /* Z has type Unnamed1, so add the constraint Unnamed1 -> A       */
  unify(Z,Y).    /* From the constrant above, we see that Z has type A
                    The constraint A -> B is invalid (named variables cannot be bound)
                    Type-checking fails                                            */
```

FIGURE 3.13: *An example of type-checking that requires type variable bindings to be tracked whilst predicate calls are type-checked.*

3.6.1 Module structure for the execute library

The complexity in the module structure for the execute library (see Figure 3.14) arises from wanting mutually recursive functions to pattern-match on the instruction being executed, and to implement the instruction. A simple solution is placing the pattern-matching function and the instruction implementation function in the same file, with explicit mutual recursion, but this approach leads to a very large file full of mutually recursive functions.

Instead, I used mutual recursion between two modules—`DriverImpl` and `InstructionImplementationsImpl`—to replace the mutual recursion between functions. This allows the function definitions to be split across the two files `execute.ml` and `instructionImplementations-Functor.ml`, inside the two mutually recursive modules. These two modules are instantiated together within the functor `DriverAndFunctions`, and this functor is instantiated in turn by providing a module `State` containing the state needed by the `DriverImpl` at run-time (for example, `State` contains the instructions to be executed).

The use of a `State` module as an argument to `DriverAndFunctions` means that the immutable state needed by the driver function is in-scope over the whole function. A more traditional approach would supply this state as a function argument to the driver, but this would require the state to be passed between `DriverImpl` and `InstructionImplementationsImpl` with every function call.

3.6.2 Testing

The code structure enables testing at the level of each library and across the whole project. The `test` library contains 84 end-to-end tests that use all of the other libraries. These tests check that the different components interact correctly to form a correct overall Prolog implementation. There are also individual tests for the `typecheck`, `code_generation`, and `execute` libraries: the directory for each of these libraries contains a sub-directory `test/` with tests for the individual

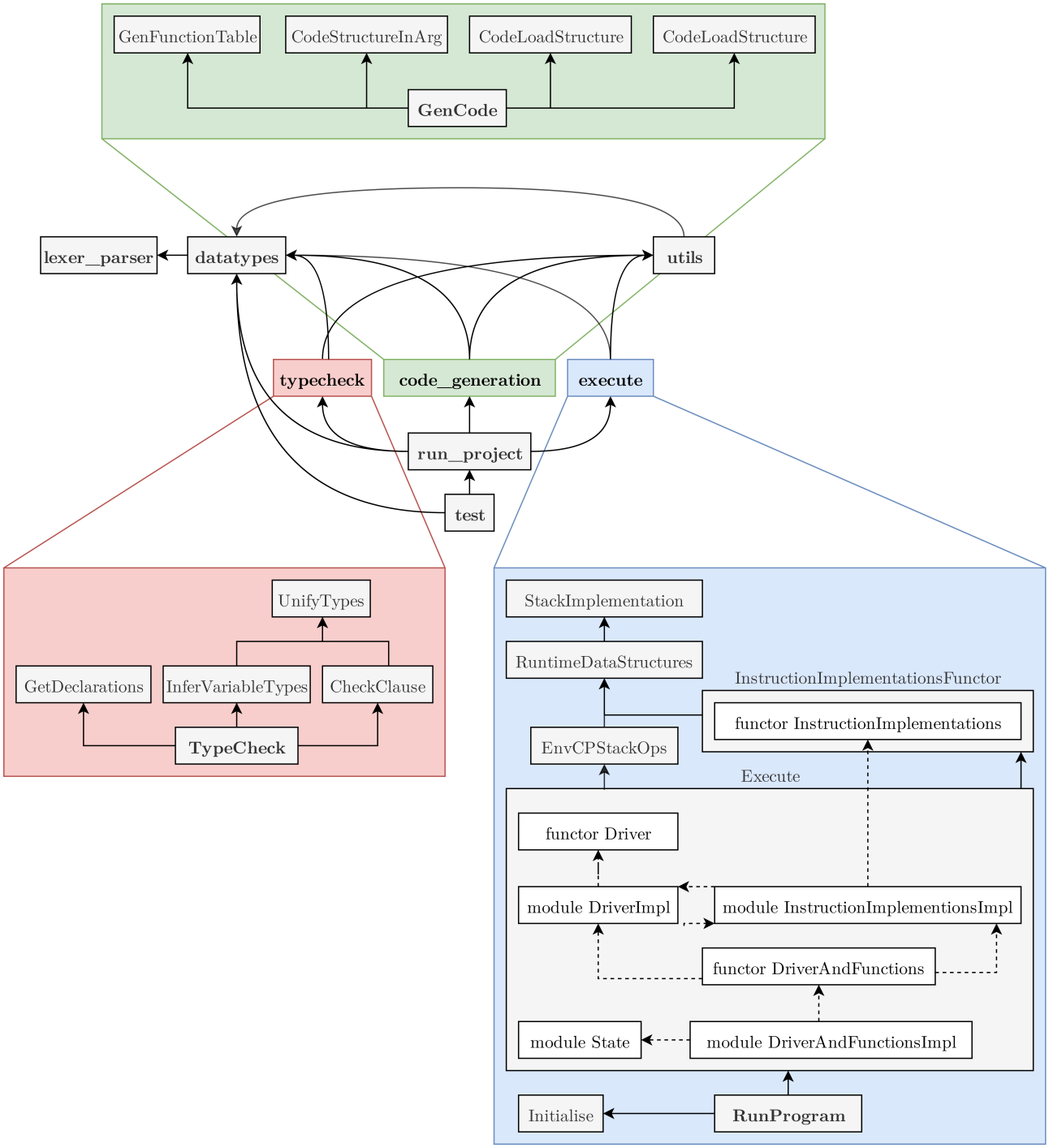


FIGURE 3.14: *Dependencies in the project. The central graph shows dependencies between libraries in the project, and the coloured boxes show dependencies between modules within the key libraries. Each file corresponds to a module, but files may also define modules explicitly; modules defined explicitly are shown as white boxes within the file's grey box.*

component. These tests are written in addition to the end-to-end tests because they make it easier to find the region of code causing a bug, and because some bugs are more easily exposed by inspecting intermediate values from the compilation process.

3.7 Summary

- A lexer and parser for Prolog were implemented using `ocamllex` and `menhir` (§3.1).
- An abstract machine (§3.2) was implemented to execute a WAM-based instruction set (§3.3). A translator was written to convert a parse tree to this instruction set (§3.4).
- The instruction set contains four main types of instructions:
  - *Control* instructions to handle the calling of predicates.
  - *Get* and *StructGet* instructions to fetch the actual parameters of a predicate.
  - *Put* and *Set* instructions to set the formal parameters for a predicate call.
  - *Is-expression* instructions to perform arithmetic.
- The abstract machine uses four key memory areas:
  - The *environment stack* stores environments containing the state associated with the invocation of a clause.
  - The *choice-point stack* stores choice points containing the state needed to revert the execution of a program on backtracking.
  - The *trail stack* stores pointers to variables that have been bound during unification and may need to be unbound on backtracking.
  - The *heap* stores variable values pointed to from the environment stack.

  The environment stack and choice-point stack store pointers into each other (§3.2.2); these are used to implement backtracking and cut.
- A polymorphic type system similar to that of ML—the Mycroft-O'Keefe type system—was implemented, and can be used to detect programmer errors (§3.5).

Chapter 4

Evaluation

In this section, the success criteria are shown to be achieved (§4.1) and Pholog's performance is compared to other implementations of Prolog (§4.3). This is followed by a discussion of how garbage collection behaviour can affect Pholog's execution speed (§4.4), and a demonstration of how last-call optimisation reduces execution time and memory usage (§4.5). The type system is then evaluated in terms of utility and correctness (§4.6).

4.1 Success criterion

The project was a success, all of the original requirements were satisfied:

- ✓ A lexer and parser were implemented.
- ✓ A translator from the parse tree to instruction sequence was implemented.
- ✓ A WAM-like abstract machine was implemented to interpret the instruction sequence.

These components combine to give a working Prolog implementation (§4.1.1). I also achieved the extension goals of adding a type system and performing last-call optimisation.

4.1.1 Implementation correctness

The correctness of my Prolog implementation was verified with tests for each individual component, as well as a large set of system-wide tests. A total of 84 system-wide tests were used, including complex problems such as 12-Queens, and edge cases such as correctly handling variable aliasing. A test case was written to prevent regression of each bug found during development.

4.2 Test environment

All performance measurements were taken on an unloaded Dell XPS 15, with specifications:

- 16GB RAM
- Intel Core i7-7700HQ processor
- Running Ubuntu 18.04.1 LTS

All tests were repeated five times, and error bars show $\pm k\sigma$ (where k is determined visually).

4.3 Speed of execution for benchmarks

In this section, my interpreter is compared to SWI-Prolog and GNU-Prolog in terms of time taken to execute benchmark programs. SWI-Prolog is a popular, open-source WAM-based bytecode interpreter [21] and therefore is expected to have similar performance to my implementation, although SWI-Prolog will likely perform better since it is written in C and has had many more hours of work put into it. Unlike SWI-Prolog, GNU-Prolog compiles to a native binary: it does not use an interpreter [5]. For this reason, GNU-Prolog is expected to outperform the other Prolog systems. It is interesting to compare against GNU-Prolog to see how Pholog's performance could be improved, and GNU-Prolog has also been a useful sanity check for the benchmarks where my implementation significantly out-performs SWI-Prolog.

Figure 4.1 shows the performance of the three implementations for various benchmark programs. These were chosen to represent a rage of different types of program (there are benchmarks

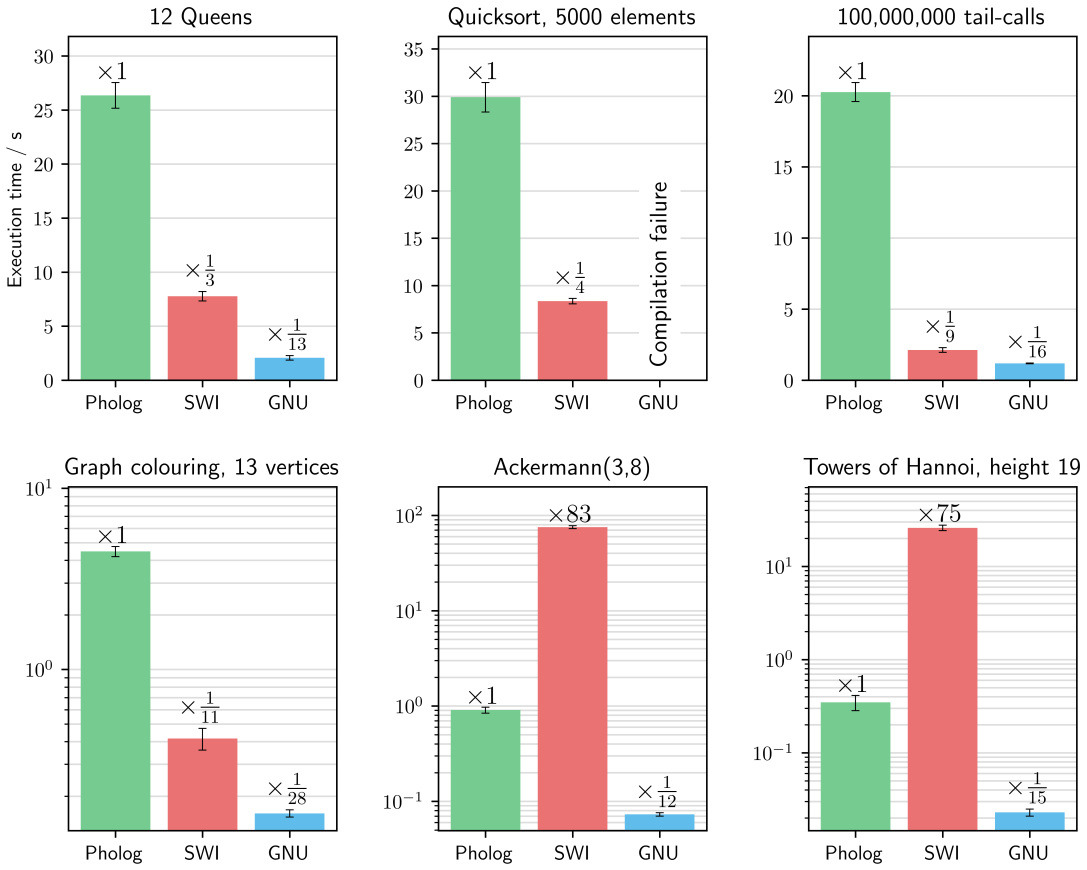


FIGURE 4.1: *A comparison of the time taken to execute sample Prolog programs. The error bars show* $\pm 5\sigma$.

with and without arithmetic, with and without amenability to last-call optimisation, etc). As expected, GNU-Prolog has the best performance for every benchmark that it could compile, although it was unable to compile the list of 5,000 elements used in the quicksort benchmark (compilation would fail without specifying any error when the input list was too long).

My implementation ranged between being 15 and 28 times slower than GNU-Prolog, and between 11 times slower and 83 times faster than SWI-Prolog. It appears that SWI-Prolog outperforms my implementation in most cases, but SWI-Prolog suffers from significant worst-cases that I avoid. These are characterised in the next section.

4.3.1 Benchmarks in which SWI-Prolog performs poorly

There are two benchmarks where my implementation outperforms SWI-Prolog: Ackermann, and Hannoi. In general, my implementation outperforms SWI-Prolog for programs that allocate structures in environments that are part of a large stack.

Listing 4.1 gives a small example of such a program, and Figure 4.2 graphs the execution time of this program as a function of the number of predicate calls. This example program is not tail-recursive, and therefore the maximum number of live environments is proportional to the number of predicate calls. Figure 4.2 shows that SWI-Prolog experiences increasing slowdown as the stack size grows: the execution time appears to be quadratic in the number of iterations of the predicate. Similar issues are seen when increasing the number of structures allocated inside the predicate, or the size of the single allocated structure. Overall, SWI-Prolog has poor

```
any(A).

test(0).
test(N) :- any(f(a)),   /* Allocate a structure                              */
           N1 is N - 1,
           test(N1),    /* Perform a recursive call                         */
           any(N1).     /* Prevent last-call optimisation of the recusive call */

?- test(x).
```

LISTING 4.1: *Example program to show the poor scaling of SWI-Prolog when allocating structures with a large stack. The value of x is varied in Figure 4.2.*

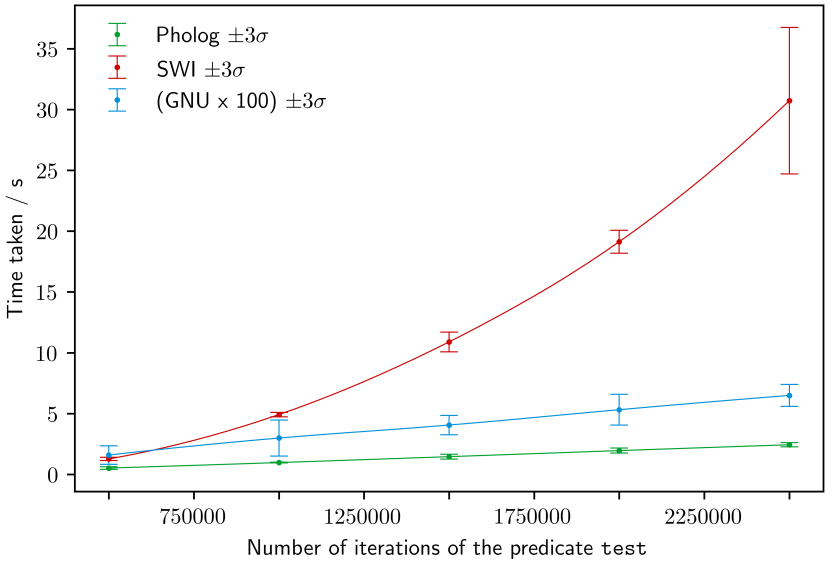


FIGURE 4.2: *Scaling of the `test` predicate from Listing 4.1; the x-axis is the value of x. The line connecting points for SWI-Prolog uses quadratic interpolation.*

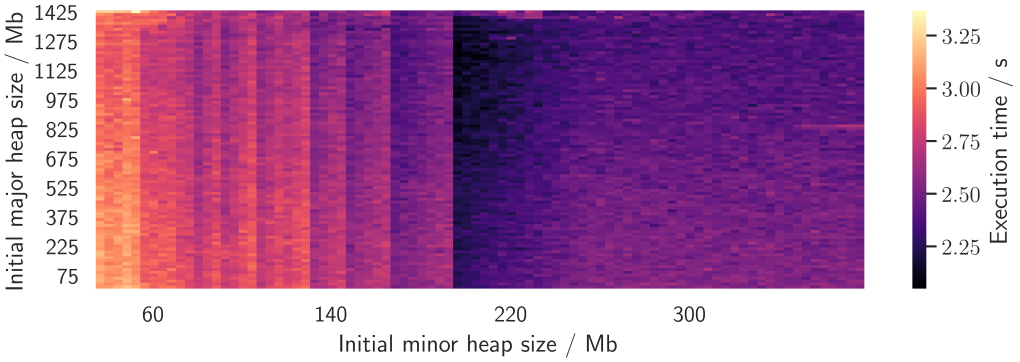
---

performance when allocating structures on a large stack; these issues are not experienced by my implementation, and nor by GNU-Prolog.

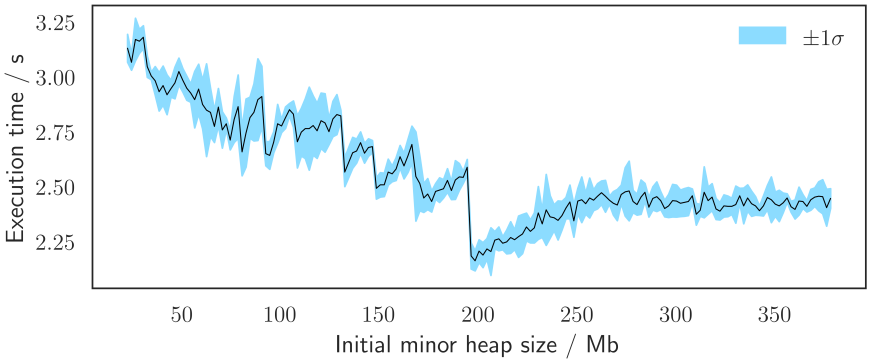
The cause of the performance difference is likely a problem with SWI-Prolog, rather than any advantage of my implementation. This is because GNU-Prolog has the same linear scaling behaviour as my implementation, but has an implementation more similar to SWI-Prolog (GNU-Prolog does not share Pholog's divergences from the WAM). SWI-Prolog appears to be performing work linear in the stack size when allocating structures, whereas the other implementations perform constant work. I cannot tell why this is happening without taking a close look at the SWI-Prolog source code. One possible explanation is that the SWI-Prolog heap becomes too small and is expanded by copying the entire stack and moving it a fixed distance, but this is purely speculation.

4.4 Performance of garbage collection

My implementation relies on garbage collection for memory deallocation, unlike other Prolog implementations. The behaviour of the OCaml garbage collector can be controlled with the



(A) *Execution time as a function of heap parameters*



(B) *Slice of* (A) *at major heap size = 525 MB*

FIGURE 4.3: *The effect of minor and major heap size on the execution time of the Ackermann benchmark. Discontinuities occur when additional rounds of garbage collections are triggered.*

---

environment variable `OCAMLRUNPARAM`: this was used to test the influence of garbage collection settings on program run-time. OCaml uses two heaps: the *minor* heap, a fixed-size heap where blocks are initially allocated; and the *major* heap, a variable-size heap for longer-lived blocks [14]. Figure 4.3 shows how the initial sizes of the minor and major heap affect the time taken to execute the Ackermann benchmark.

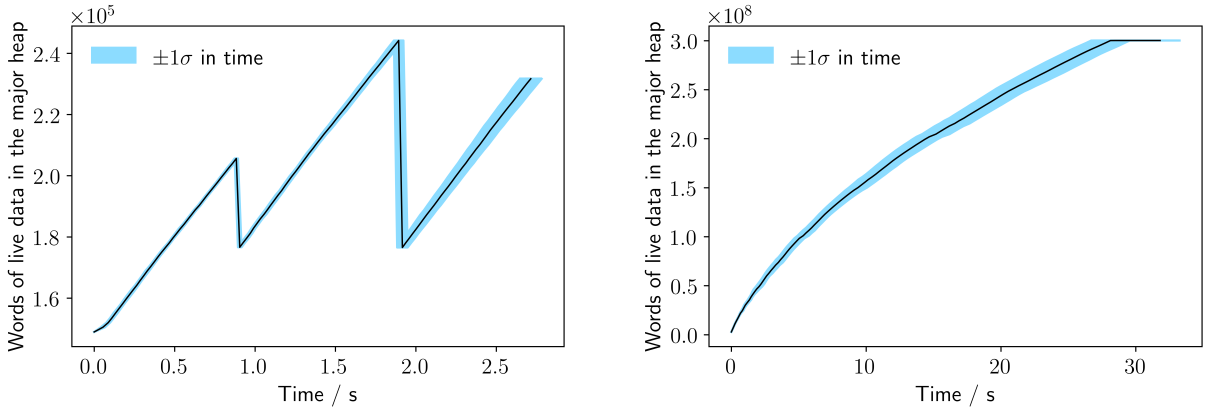
Figure 4.3 demonstrates the importance of garbage collection parameters for program performance: the best performance achieved across the range of heap sizes tested is 50% faster than worst performance. Performance is affected by the initial minor heap size much more significantly than it by the initial major heap size; this is because the major heap can be expanded dynamically, but the minor heap is a fixed size. This shows that getting the best performance from my interpreter requires tuning the minor heap size per-program, and also suggests that the overhead from expanding the major heap is not significant.

SWI-Prolog and GNU-Prolog also require manual tuning to parameters controlling the amount of memory used for the different memory regions of the WAM (see Figure 3.2); these implementation will fail with a overflow error if not started with large enough memory regions. One advantage of my implementation is that there is no need to specify a fixed maximum size for memory regions, avoiding this failure case.

For the benchmarks in Section 4.3, the initial minor and major heap sizes were set to 3GB. This seems large, but it is conservative in comparison to SWI and GNU Prolog, which were configured to use up to 16GB of memory to prevent stack-overflow errors.

```
iter(0).
iter(N) :- N1 is N - 1, iter(N1).

?- iter(10000000)
```

LISTING 4.2: *A Prolog program expected to benefit significantly from last-call optimisation.*



(A) *With last-call optimisation* (B) *Without last-call optimisation*

FIGURE 4.4: *A comparison of the memory profile of the program in Listing 4.2 with and without last-call optimisation.*

4.5 Last-call optimisation

The program shown in Listing 4.2 is constructed to determine the best-case effectiveness of last-call optimisation: the program makes 10,000,000 tail-recursive calls, and only performs one subtraction per call. The time taken to execute this program with last-call optimisation is 2.876 seconds (with $\sigma = 0.06$), and the time taken without is 10.4 seconds (with $\sigma = 0.4$): the optimisation gives a four times improvement in execution time. Last-call optimisation improves execution time because the major heap must be expanded fewer times, and because less data must be copied from the minor heap to the major heap.

Figure 4.2 shows the number of words of live data in OCaml's major heap over time for this program, with and without last-call optimisation. Sub-figure (A) shows sharp peaks in the amount of live data: each of these peaks is caused by a garbage collection, when many environments can be garbage collected because last-call optimisation has removed all pointers to these environments. In sub-figure (B), the amount of live data increases monotonically to a maximum of about 1000 times the maximum of (A); this occurs because the garbage collector cannot free the memory associated with any environments until the program has completed execution.

The total amount of memory allocated by the interpreter is the same whether or not last-call optimisation is used, but this is not reflected in Figure 4.4: it appears that significantly more memory is allocated without last-call optimisation. This is because the figure does not include memory allocated on the minor heap: with last-call optimisation, many environments never need to be copied from the minor heap to the major heap.

```
type list A = cons(A, list(A)), eol.
type tree A = tree(A, tree(A), tree(A)), leaf.
type direction = left, right.

pred unify(B,B).
unify(X,X).

pred dfs(C, tree(C), list(direction)).
dfs(V,tree(V,T1,T2),eol).
dfs(V,tree(R,T1,T2),Path1) :- dfs(V,T1,Path2), unify(P1,cons(left,Path2)).
dfs(V,tree(R,T1,T2),Path1) :- dfs(V,T2,Path2), unify(P1,cons(right,Path2)).
```

(A) *A typed Prolog program*

Variable	Inferred type
X	TypeVar(B)
T1	TypeCons(tree, [TypeVar(C)])
T2	TypeCons(tree, [TypeVar(C)])
V	TypeVar(C)
Path1	TypeCons(list, [TypeCons(direction,[])])
Path2	TypeCons(list, [TypeCons(direction,[])])
R	TypeVar(C)

(B) *Inferred variable types from* (A)

FIGURE 4.5: *A typed Prolog program, and the types inferred for its variables.*

4.6 Type system

4.6.1 Evaluation of correctness

Mycroft and O'Keefe proved that a well-typed program (under their type system) cannot experience run-time type errors [15]. This section justifies that my type system correctly implements the rules of the Mycroft-O'Keefe type system (Figure 2.4, page 8) using a series of examples.

Variable type inference

The type system does not use manual type declarations for the types of variables; instead, the types of variables are inferred. This inference has been tested by adding logging to the interpreter to observe the type inferences that are made. Figure 4.5 gives a Prolog program and shows the type inferences reported by this logging. Pholog correctly infers the type of all variables including `Path2`, which does not have its type specified by the predicate type declaration.

Definitional generality

The type rules enforce that the definition of a predicate should be as general as its type signature: for example, a predicate declared with to have `A` cannot be defined by a clause only accepting integers. Predicate calls may, however, be performed with any instantiation of the predicate's type. A number of test cases have been constructed for this behaviour, and are presented in Table 4.1. The implementation passes all of these tests.

Nesting types

Types can be defined in terms of other types, potentially recursively. For example, the program in Listing 4.3 uses the type definition `type map A,B = mkmap(list(pair(A,B)))`. This type definition uses the types `list` and `pair`, and is correctly accepted by the type system.

Program	Typeable?	Explanation
`pred p(A).` `p(1).`	✗	The type `A` is more general than the type `int`, but the clause definition should be as general as its type declaration.

`pred p(A).`
`p(X).`
`?- p(1)`
	✓	The type `int` is a valid instantiation of the type `A`.

`pred p(A,B).` `p(X,X).`	✗	The type variables `A` and `B` cannot be unified; this ensures that the clause is as general as its type declaration.

`pred p(A,B).`
`p(X,Y).`
`?- p(Z,Z)`
	✓	The type `Z` is a valid instantiation of the types `A` and `B`: `A` and `B` can be unified for a predicate call.

`pred unify(A,A).`
`unify(X,X).`
 	
`pred p(A,B).`
`p(X,Y) :-`
`  unify(X,Z),`
`  unify(Z,Y).`
	✗	The variable `Z` needs to have the same type as `X` and `Y`, but the types of `X` and `Y` cannot be unified.

TABLE 4.1: *Test cases to demonstrate definitional generality of the type system.*

```
type list A = cons(A, list(A)), eol.      type pair A, B = mkpair(A,B).
type map A, B = mkmap(list(pair(A,B))).   type opt A = some(A), none.

pred mapLookup(A, map(A,B), opt(B)).
mapLookup(V, mkmap(eol), none).
mapLookup(V, mkmap(cons(mkpair(V,K),Tail)), some(K)).
mapLookup(V, mkmap(cons(H,T)),Result) :- mapLookup(V, mkmap(T),Result).
```

LISTING 4.3: *Typed Prolog program using types defined in terms of other types.*

4.6.2 Errors that the type system can catch

The type system is motivated by catching programmer errors. One potential programmer error that could be caught is using the wrong program variable. For example, the program below is intended to look-up a value from a list of key-value pairs, but a key is mistakenly returned instead of a value.

```
type list A = cons(A, list(A)), eol.      type pair A, B = mkpair(A,B).

pred lookup(A, list(pair(A,B)), B).
lookup(K, cons(mkpair(K,V), Tail), K).
lookup(L, cons(H,T), Result) :- lookup(K, T, Result).
```

This program error will be caught by the type system: `K` has type `A`, and cannot be used in a position requiring it to have type `B` due to the definitional generality constraint.

Another error that can be caught is re-ordering of predicate arguments. The example program

	Implementation language	Compilation target	Clause indexing	Static type-checker
SWI Prolog	C	Bytecode	✓	✗
GNU Prolog	C	Native code, bytecode	✓	✗
Ciao	Ciao (Prolog variant)	Native code, bytecode, C	✓	✓
This project	OCaml	Bytecode	✗	✓

TABLE 4.2: *Comparison to existing Prolog implementations.*

below gives a predicate to calculate the length of a list, but the arguments in the recursive call have been swapped in order.

```
type list A = cons(A, list(A)), eol.

pred length(list(A), int).
length(eol, 0).
length(cons(H,T), Res) :- length(Res1, T), Res is Res1 + 1.
```

This error is caught by the type system in the variable type inference pass. This pass unifies all of the types that a variable must satisfy, but a `list` type can never be unified with an `int`.

The type system also detects most errors occurring due to typos in structure names. For example, we might have the type declaration `type pair A, B = mkpair(A,B)` but could use the structure `mkpai` instead of `mkpair` in some clause. This error would be detected by the type system when it tried (and failed) to determine the type of the structure '`mkpair`'. Catching this class of errors in Prolog is significant because structures in untyped Prolog have an infinite namespace; if you take a Prolog program and change one instance of the name of a structure then the program will remain syntactically valid. Adding a type system enables these renaming errors to be detected.

4.7 Comparison to other Prolog implementations

Figure 4.2 gives a comparison between the features of my Prolog implementation and those of others. My implementation has a number of advantages over others:

- I observed bugs in both GNU-Prolog and SWI-Prolog that would be unlikely to occur in my implementation. When benchmarking GNU-Prolog, I experienced problems that would be unlikely to occur using an OCaml-based compiler: for example, compilation failure due to a segmentation fault. I also observed SWI-Prolog to have unexpected quadratic time complexity for programs allocating memory with a large environment stack (§4.3.1). Because Pholog uses OCaml's runtime for memory allocation, these of sorts problems would be unlikely to arise.
- It is rare for Prolog systems to include a type system: SWI-Prolog and GNU-Prolog do not perform static type checking, although implementations with optional type systems such as Ciao exist [6]. As shown in §4.6.2, adding a type system enables Pholog to detect program errors before runtime.
- By switching to using OCaml's garbage collector for memory management, Pholog is able

to use an environment stack representation that makes environment and choice point deallocation simple. Environment deallocation is more complex when using the traditional local stack (of choice points and environments) because environments cannot be deallocated whilst they are needed for a choice point, even if the environment's clause has finished execution. *Environment protection* must be implemented to avoid environments that are needed for choice points being deallocated.

My implementation avoids this complexity by keeping separate stacks for choice points and environments, and maintaining the invariant that there exists a pointer to every environment needed for the program. The deallocation of environments can therefore be performed by the garbage collector.

- Prolog systems are commonly implemented in C, whereas my implementation uses OCaml. This switch makes project development and maintenance easier in general, thanks to features such as static type-checking and automatic memory management, and increases confidence that the implementation is correct.

However, there are also significant disadvantages associated with my implementation:

- My implementation is not as performant as SWI-Prolog in majority of cases (§4.3), even though both implementations interpret WAM-like bytecode. This is partly due to other Prolog implementations benefiting from many more total hours of development, but my switch to using OCaml—particularly the switch to garbage collecting the interpreter's runtime stacks—is likely to have lead to performance losses.
- My implementation does not support compilation to native machine code. This limits performance: when compiled to native code, GNU-Prolog significantly outperforms my implementation.
- Whilst my implementation includes all of the core logical features of Prolog—as well as arithmetic, cut and last-call optimisation—it lacks many features of larger Prolog implementations. My implementation lacks common library predicates, does not include a debugger, and does not have an interactive top-level.

In summary, my implementation benefits from a switch to OCaml. This reduces the chance of bugs in the implementation, and enables a simplification of runtime memory layout. However, other Prolog implementations are typically more performant and have more features.

4.8 Summary

- All the core requirements for Pholog (§2.7) were achieved, as well as significant extensions (type-checking and last-call optimisation). The correctness of the implementation was verified using system-wide and component-specific tests (§4.1).
- Pholog achieves speed comparable to SWI-Prolog (§4.3).
- Correctness of Pholog's implementation of the Mycroft-O'Keefe type system was demonstrated with example programs that correspond to difficult cases for the type-checker (§4.6.1).
- Utility of the type system was evaluated by showing program errors that it could catch (§4.6.2).
- The advantages and disadvantages of Pholog over other Prolog implementations have been discussed (§4.7).

Chapter 5

Conclusions

5.1 Work completed

The project was successful: I implemented a Prolog system (*Pholog*) meeting all success criteria, and added a type system as well as last-call optimisation for extensions. Three main components were implemented: a lexer and parser, a translator to a bytecode-style instruction set, and an abstract machine to execute this instruction set. The abstract machine was inspired by the Warren Abstract Machine, a common target for Prolog implementations. Key differences between my abstract machine and the WAM include the introduction of new instructions for arithmetic and cut, as well as a change in memory layout to better exploit the use of a garbage-collected language. Pholog has some advantages over other Prolog implementations, including:

- Pholog uses a simpler memory layout: the traditional stack of choice points and environments is separated into two stacks. This is possible because Pholog uses garbage collection for memory management.
- Pholog is written in OCaml: other systems commonly use C. OCaml's static type checking and automatic memory management help prevent bugs in the implementation. I show that switching to OCaml does not significantly affect performance (§4.3).
- Pholog includes an optional type system: this can detect many programmer errors in Prolog programs (§4.6.2), but most popular Prolog systems do not include this feature.

5.2 Lessons learned

During this project, I learnt a great deal about Prolog and the implementation of compilers and interpreters. I also learnt to use a new programming language—OCaml—and to exploit the features of this language to present a Prolog implementation with a different set of trade-offs to standard C-based implementations. If I were to repeat the project, I would change the project schedule. I would include arithmetic and cut as planned extensions, and would plan to continue project development for longer into the Lent term.

5.3 Future work

Pholog would benefit from the addition of a *mode system* [4]. This is similar to a type system, but characterises whether terms are ground (containing no free variables), partially instantiated (containing free variables as within an instantiated structure), or unbound (consisting only of an unbound variable). A mode system can detect programmer errors where terms do not have the specified instantiation properties. Performance could also be improved using the mode system. For example, ground terms are immutable so could be duplicated and collapsed to remove pointer indirection.

Since around 2005, improvements in single-threaded performance have been limited: instead, performance gains have been achieved via increased parallelism. Currently, Pholog is single-threaded so cannot exploit the increasing numbers of cores present on modern processors. Performance could be improved by parallelising Pholog. There are two main ways to automatically parallelise a Prolog program: *or-parallelism* and *and-parallelism* [16]. Both of these could be added to Pholog, although or-parallelism is typically easier to implement. One difficulty is that OCaml does not support shared-memory thread-level parallelism [17], meaning that parallelising Pholog would either require a switch of implementation language or a modification of OCaml.

Bibliography

[1] Lexer and parser generators (ocamllex, ocamlyacc). https://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html (accessed April 20, 2019). OCaml Documentation.

[2] OUnit. http://ounit.forge.ocamlcore.org/ (accessed April 20, 2019). OCaml unit testing framework.

[3] Package Core documentation. https://ocaml.janestreet.com/ocaml-core/latest/doc/core/ (accessed April 22, 2019).

[4] Saumya K. Debray and David S. Warren. Automatic mode inference for logic programs. *The Journal of Logic Programming*, 5(3):207 – 229, 1988.

[5] Daniel Diaz. The GNU prolog website. http://www.gprolog.org/ (accessed April 20, 2019).

[6] M. V. Hermenegildo, F. Bueno, M. Carro, P. Lípez-García, E. Mera, J. F. Morales, and G. Puebla. An overview of ciao and its design philosophy. *Theory Pract. Log. Program.*, 12(1-2):219–252, January 2012.

[7] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: An interactive tutorial. *SIGMOD*, pages 1213 – 1216, 2011.

[8] Robert Kowalski. *Predicate Logic as Programming Language*, 01 1974.

[9] Robert Kowalski and Donald Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 1971.

[10] Robert A. Kowalski. The early years of logic programming. *Communications of the ACM*, 2002.

[11] T. L. Lakshman and Uday S. Reddy. Typed prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In Vijay A. Saraswat and Kazunori Ueda, editors, *ISLP*, pages 202–217. MIT Press, 1991.

[12] Adam Lally and Paul Fodor. Natural language processing with Prolog in the IBM Watson system. *Association for Logic Programming*, 2011.

[13] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. The Java Virtual Machine Specification Java SE 11 Edition, 2018.

[14] Anil Madhavapeddy, Jason Hickey, and Yaron Minsky. *Real World OCaml: Functional Programming for the Masses*. O'Reilly Media, 2013.

[15] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for prolog. *Artificial Intelligence*, 23(3):295 – 307, 1984.

[16] João Santos and Ricardo Rocha. On the implementation of an or-parallel prolog system for clusters of multicores. *CoRR*, abs/1608.01499, 2016.

[17] KC Sivaramakrishnan. State of multicore OCaml. OCaml development meeting, Paris, 2018.

[18] Y. Smaragdakis and M. Bravenboer. Using datalog for fast and easy program analysis. *Datalog 2.0 Workshop*, 2010.

[19] M. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733 – 742, 1976.

[20] David H. D. Warren. An abstract Prolog instruction set. *SRI international*, 1983.

[21] Jan Wielemaker. Swi-prolog 6.1. http://www.swi-prolog.org/download/devel/doc/SWI-Prolog-6.1.5.pdf (accessed April 20, 2019). Reference Manual.

Appendix A

Example Program

An example program to be compiled

```
type list A = c(A, list(A)), eol.

pred leq(int,int).
leq(1,1).
leq(1,2).
leq(1,3).
leq(2,2).
leq(2,3).
leq(3,3).

pred take(list(A),A,list(A)).
take(c(H,T),H,T).
take(c(H,T),R,c(H,S)) :- take(T,R,S).

pred perm(list(A),list(A)).
perm(eol,eol).
perm(X,c(H,T)) :- take(X,H,R), perm(R,T).

pred sorted(list(int)).
sorted(eol).
sorted(c(H,eol)).
sorted(c(H1,c(H2,T))) :- leq(H1,H2), sorted(c(H2,T)).

pred terribleSort(list(int),list(int)).
terribleSort(X,Y) :- perm(X,Y), sorted(Y).


?- terribleSort(c(3,c(1,c(1,c(2,eol)))),A)
```

As expected, this program successfully type-checks.

Instructions generated for the program

```
0. (Allocate 1)
1. (PutStructTemp ((0, 0), (Temp 3)))
2. (PutStructTemp ((1, 2), (Temp 2)))
3. (SetInt 2)
4. (SetValue (T (Temp 3)))
5. (PutStructTemp ((1, 2), (Temp 1)))
6. (SetInt 1)
7. (SetValue (T (Temp 2)))
8. (PutStructTemp ((1, 2), (Temp 0)))
9. (SetInt 1)
10. (SetValue (T (Temp 1)))
11. (PutStructArg ((1, 2), (Arg 0)))
12. (SetInt 3)
13. (SetValue (T (Temp 0)))
14. (PutVariable ((E (Env 0)), (Arg 1)))
15. (Call (PositionF 115))
16. Finish
17. (TryMeElse (PositionC 22))
18. (Allocate 0)
19. (GetInt ((Arg 0), 1))
20. (GetInt ((Arg 1), 1))
21. Deallocate
22. (RetryMeElse (PositionC 27))
23. (Allocate 0)
24. (GetInt ((Arg 0), 1))
25. (GetInt ((Arg 1), 2))
26. Deallocate
27. (RetryMeElse (PositionC 32))
28. (Allocate 0)
29. (GetInt ((Arg 0), 1))
30. (GetInt ((Arg 1), 3))
31. Deallocate
32. (RetryMeElse (PositionC 37))
33. (Allocate 0)
34. (GetInt ((Arg 0), 2))
35. (GetInt ((Arg 1), 2))
36. Deallocate
37. (RetryMeElse (PositionC 42))
```

```
38. (Allocate 0)
39. (GetInt ((Arg 0), 2))
40. (GetInt ((Arg 1), 3))
41. Deallocate
42. TrustMe
43. (Allocate 0)
44. (GetInt ((Arg 0), 3))
45. (GetInt ((Arg 1), 3))
46. Deallocate
47. (TryMeElse (PositionC 52))
48. (Allocate 0)
49. (GetStructArg ((0, 0), (Arg 0)))
50. (GetStructArg ((0, 0), (Arg 1)))
51. Deallocate
52. TrustMe
53. (Allocate 4)
54. (GetVariable ((E (Env 0)), (Arg 0)))
55. (GetStructArg ((1, 2), (Arg 1)))
56. (StructGetVariable (E (Env 1)))
57. (StructGetVariable (E (Env 2)))
58. (PutValue ((E (Env 0)), (Arg 0)))
59. (PutValue ((E (Env 1)), (Arg 1)))
60. (PutVariable ((E (Env 3)), (Arg 2)))
61. (Call (PositionF 93))
62. (PutValue ((E (Env 3)), (Arg 0)))
63. (PutValue ((E (Env 2)), (Arg 1)))
64. (Call (PositionF 47))
65. Deallocate
66. (TryMeElse (PositionC 70))
67. (Allocate 0)
68. (GetStructArg ((0, 0), (Arg 0)))
69. Deallocate
70. (RetryMeElse (PositionC 77))
71. (Allocate 1)
72. (GetStructArg ((1, 2), (Arg 0)))
73. (StructGetVariable (E (Env 0)))
74. (StructGetVariable (T (Temp 1)))
75. (GetStructTemp ((0, 0), (Temp 1)))
76. Deallocate
77. TrustMe
78. (Allocate 3)
79. (GetStructArg ((1, 2), (Arg 0)))
80. (StructGetVariable (E (Env 0)))
81. (StructGetVariable (T (Temp 1)))
82. (GetStructTemp ((1, 2), (Temp 1)))
83. (StructGetVariable (E (Env 1)))
84. (StructGetVariable (E (Env 2)))
85. (PutValue ((E (Env 0)), (Arg 0)))
86. (PutValue ((E (Env 1)), (Arg 1)))
87. (Call (PositionF 17))
88. (PutStructArg ((1, 2), (Arg 0)))
89. (SetValue (E (Env 1)))
90. (SetValue (E (Env 2)))
91. (Call (PositionF 66))
92. Deallocate
93. (TryMeElse (PositionC 101))
94. (Allocate 2)
95. (GetStructArg ((1, 2), (Arg 0)))
96. (StructGetVariable (E (Env 0)))
97. (StructGetVariable (E (Env 1)))
98. (GetValue ((E (Env 0)), (Arg 1)))
99. (GetValue ((E (Env 1)), (Arg 2)))
100. Deallocate
101. TrustMe
102. (Allocate 4)
103. (GetStructArg ((1, 2), (Arg 0)))
104. (StructGetVariable (E (Env 0)))
105. (StructGetVariable (E (Env 1)))
106. (GetVariable ((E (Env 2)), (Arg 1)))
107. (GetStructArg ((1, 2), (Arg 2)))
108. (StructGetValue (E (Env 0)))
109. (StructGetVariable (E (Env 3)))
110. (PutValue ((E (Env 1)), (Arg 0)))
111. (PutValue ((E (Env 2)), (Arg 1)))
112. (PutValue ((E (Env 3)), (Arg 2)))
113. (Call (PositionF 93))
114. Deallocate
115. (Allocate 2)
116. (GetVariable ((E (Env 0)), (Arg 0)))
117. (GetVariable ((E (Env 1)), (Arg 1)))
118. (PutValue ((E (Env 0)), (Arg 0)))
119. (PutValue ((E (Env 1)), (Arg 1)))
120. (Call (PositionF 47))
121. (PutValue ((E (Env 1)), (Arg 0)))
122. (Call (PositionF 66))
123. Deallocate
```

Appendix B

Extract from the Interpreter

Extract from the driver function for the interpreter

```
let rec execute compState : 'a option =
  let cp = compState.cp
  in let choicePoints = compState.choicePoints
  in let envStack = compState.envStack
  in match ((Array.get State.instructions cp), choicePoints, envStack) with

  | TryMeElse(PositionC(n)), _, _ ->
    (* Create a new choice point *)

    let newChoicePoint = {
      nextOptionPointer = n;
      stack = envStack;
      arguments = (Array.copy compState.arguments);
      returnAddr = compState.returnAddress;
      returnCps = compState.returnCps;
      returnTrailPoint = compState.returnTrailPoint;
    }
    in compState.trail <- createNew Label ~prev:compState.trail;
    compState.choicePoints <- Some({prev = choicePoints; value = newChoicePoint});
    compState.cp <- compState.cp + 1 ;
    execute compState

...

  | GetVariable(position, Arg(a)), _, Some(env) ->
    (* Load an argument into a fresh variable *)

    let loadedVar = Array.get compState.arguments a
    in (match position with
        | T(Temp(e)) -> Array.set compState.temps e loadedVar
        | E(Env(e)) -> Array.set env.value.vars e loadedVar
      );
    compState.cp <- compState.cp + 1 ;
    execute compState

...

  | GetInt(Arg(a),n1),_,_ ->
    (* Unify an argument with an integer *)

    let loadedArg = Array.get compState.arguments a
    (* Call helper function in module InstructionImplementations *)
    in I.getinta loadedArg n1 compState
```

Appendix C

Project Proposal

My project proposal is given on the next page.

Computer Science Tripos: Part II Project Proposal

An Implementation of Prolog

Candidate 2398E
(college removed)
October 2018

Project Originator:	Candidate 2398E	**Project Supervisor:**	Prof. Alan Mycroft
Director of Studies:	(name removed)	**Overseers:**	Prof. Andrew Pitts Dr Rafal Mantiuk

Introduction and Description of the Work

Prolog is a declarative programming language. This means that a Prolog programmer declares rules that describe the result of a program, but does not specify how the program should achieve this result. This is in contrast to imperative programming languages such as Java, where the programmer specifies how a program's state should be changed in order to achieve the result of the program. The use of a declarative language presents the interesting problem of how to evaluate a query without being told how the answer to the query should be computed. In the case of Prolog, this problem is solved by performing a graph search over the Horn clauses given in a Prolog program. This search attempts to find an assignment of values to variables that satisfies the query by repeatedly unifying a literal with the head of a clause and then trying to prove that clause, and backtracking if no more progress can be made. The query is rejected if no assignment to satisfy the query exists. Typically, a left-to-right depth-first search is performed.

The aim of this project is to implement a compiler and abstract machine for Prolog. The target for the Prolog compiler will be specialised byte-code. Designing this byte-code will form part of the project. The abstract machine implemented will interpret this byte-code to run the original Prolog program. An existing specification of an abstract machine to execute Prolog, the Warren Abstract Machine (WAM), will be used as inspiration—but I will not implement the full WAM.

The project will be restricted to a basic subset of Prolog so that it remains feasible. Some extensions that I am considering (for example type checking) will require an extension to Prolog syntax in order to provide information about program properties. The syntax I use may therefore not be a true subset of standard Prolog.

Starting Point

Prolog as a programming language

I will need a good understanding of how to program in Prolog in order to implement the Prolog compiler. I will also need to write test programs for the subset of Prolog that I have chosen to compile. These tests will help me to evaluate the correctness and performance of my compiler and abstract machine implementation.

I am studying the 50% course, which does not cover Prolog until Lent term this year. I have, therefore, studied the Prolog course over the summer.

Prolog compilation and execution

I have no prior experience with implementing Prolog interpreters or compilers. The IB 75%/II 50% Prolog course that I studied over the summer introduced me to the basics of Prolog

execution. I have also read around this area over the summer: some of the papers I read are cited in this document. I have not previously implemented any part of a Prolog compiler, interpreter, or abstract machine.

OCaml

I am intending to implement all of my project in OCaml initially, although I may add some SWI-Prolog for extension features. I have chosen OCaml because I have experience implementing lexers and parsers in ML from the compilers course, and found ML convenient for this sort of thing. OCaml will have the same advantages as ML, but is more widely used and therefore has more tools and libraries available.

I have very little prior experience with OCaml (only from the IB compilers course), so I will need to dedicate time to studying it. This time is provisioned in my project plan.

Structure of the Project

The project is made up of four main components:

1. Lexer
2. Parser
3. Translator
4. Abstract machine

I will implement the components in this order. This is because each component can be used to generate test cases for the next component. As an extension, I may add a semantic analysis phase acting on the parse tree. This would verify program properties and try to provide information so that the translator can generate more efficient code.

Implementation of the Abstract Machine

The abstract machine will be inspired by the Warren Abstract Machine (WAM) [1]. The WAM instruction set contains five main types of instruction:

- *Get* instructions, used to fetch a procedure's arguments. A procedure is a set of clauses with the same arity and function symbol.
- *Put* instructions, used to provide the arguments to a procedure.
- *Unify* instructions, used to perform unification.
- *Procedural* instructions, used for control transfer and environment allocation for procedure calls.
- *Indexing* instructions, used to link the different clauses making up a procedure.

These are interpreted using three main code areas called the local stack, heap (global stack), and trail. The local stack contains environments and choice points (information needed for backtracking), the heap contains structures and lists created by unification, and the trail contains references to variables that were bound on unification and need to be unbound on backtracking.

The WAM is the most standard Prolog implementation, and is used by SWI-Prolog. Other relevant literature includes: a paper describing a Prolog abstract machine that avoids the register usage of the WAM [2], and a general comparison of Prolog implementation techniques [3]. A lower-level adaptation of the WAM is given by Peter Van Roy, achieving improved performance [4]. Since all these papers base themselves on the WAM, I will be able to do a rough WAM implementation using some other ideas taken from the literature.

Testing

Unit tests will be written for each individual component. Components written in OCaml will likely have unit tests written with the testing framework OUnit [5]. Any components written

in SWI-Prolog will be tested using native support for unit tests as described in the SWI-Prolog documentation [6]. I will write unit tests for each component along with the implementation of the component itself.

Evaluation

I will be able to quantitatively evaluate the project in terms of:

- The run-time performance of my abstract machine compared to standard Prolog implementations such as SWI-Prolog.
- The run-time performance of my abstract machine compared to my interpreter.
- The change in code size, number of choice points, or performance, for any optimisations that I implement.

Here, 'performance' means a comparison of space and time usage for sample programs. I will be able to profile the memory behaviour of my OCaml programs using Spacetime [7]. This is a tool built into special versions of the OCaml compiler for memory profiling. SWI-Prolog also includes predicates for execution profiling [8].

Success Criterion

My success criterion is to implement the following components for a basic subset of Prolog:

- Lexer
- Parser
- Translator from parse tree to byte-code
- Abstract machine to execute this byte-code

These components should enable me to correctly execute example programs written in the grammar that I use. The components will not implement the full ISO Prolog standard, and the grammar used may include features such as type annotations not found in standard Prolog. The lexer and parser may be implemented manually or using tools such as ocamllex, ocamlyacc, or menhir.

Possible extensions

The main extensions for this project involve adding optimisations to the compiler. Some potential optimisations are:

- **Last call optimisation**
  Tail recursion can be converted to iteration to save adding a new stack frame for each tail recursive call.
- **Determinacy analysis**
  A clause is said to be determinate if it can only return one possible solution [9]. This information can be used to avoid backtracking through determinate clauses.
- **Mode analysis**
  The argument to a predicate can be assigned a mode to say that the argument is always used as input, or as output [10]. Mode information can be used to invoke special purpose unification routines that test for fewer cases and therefore can be faster.
- **Type checking**
  It is possible to define a polymorphic type system for Prolog [11]. Type information can be used to select faster, special-purpose unification routines. A type checker also has the significant advantage of potentially spotting program bugs and saving developer time.

There are many more possible optimisations I could apply, and I might also select one of these for an extension. As I implement the initial system I will learn more about the optimisations I could make, and may also find my implementation to be more suited to certain optimisations.

Timetable and Milestones

Key summary

- 16th Dec: Success criteria met
- 1st Feb: Progress report deadline
- 24th Feb: Draft dissertation completed
- 17th May: Dissertation deadline

22nd Oct – 4th Nov ***19th Oct**: Proposal submitted*

Project set-up:

- Investigate and set up an IDE for OCaml.
- Set up a backup system for the project.
- Choose a software engineering methodology to follow.
- Learn to use the testing frameworks needed for the project.

Write small programs in SWI-Prolog, to re-enforce understanding of Prolog and for potential use as benchmarks. Study OCaml using the Real World OCaml textbook [12], and write small programs in OCaml to gain familiarity with the language.

Milestones:

- Document produced to describe the development environment, backup system, software engineering methodology, and testing frameworks used.
- Example programs written in OCaml and Prolog.

5th Nov – 18th Nov

Implement a lexer and parser for the chosen subset of Prolog. I am intending to use ocamllex and ocamlyacc (described in [13]) for this.

Milestones:

- Code for lexer written and tested.
- Code for parser written and tested.
- Document produced to describe the implementation of these components.

19th Nov – 2nd Dec ***30th Nov**: Last day of Michaelmas term*

Implement an interpreter to evaluate the parse tree output by the parser. This interpreter will guide the design of the abstract machine, and may be useful as a performance benchmark.

Milestones:

- Code for interpreter written and tested.
- Document produced to describe the implementation of the interpreter.

3rd Dec – 16th Dec

Design the byte-code to represent Prolog programs, and write an abstract machine to execute this byte-code. Write the translator to convert the parse tree to byte-code.

Milestones:

- Code for abstract machine written and tested.

- Code for translator written and tested.
- Document produced to describe the byte-code format chosen.
- Document produced to describe how the abstract machine works.

17th Dec – 23rd Dec [*Slack*]

Read about possible extensions to the project. This could be research on type, mode, or determinacy analysis for Prolog. Of these options I expect type checking to be the most complex, and determinacy analysis the least complex. I will choose where to invest my time depending on how far ahead or behind the project is.

Milestones:

- Produce document to summarise research into optimisations.

24th Dec – 30th Dec [*Christmas break*]

31st Dec – 6th Jan [*Slack*]

Continue research into extensions.

Milestones:

- Add more content to the previous summary document.

7th Jan – 20th Jan [*Slack*] ***15th Jan**: First day of Lent term*

Implement extension features for the compiler.

Milestones:

- Code for extension features written and tested.
- Document produced to describe the theory and implementation of the extension features.

21st Jan – 27th Jan

Prepare for progress report and presentation.

Milestones:

- Progress report and presentation written.

28th Jan – 10th Feb ***1st Feb**: Progress report deadline*

Write a draft of the structure of the dissertation, giving the main points to be covered. Write a full version of the evaluation section of the dissertation.

Milestones:

- Draft structure of dissertation written and sent to DoS and supervisor.
- Evaluation section written and sent to DoS and supervisor.

11th Feb – 24th Feb

Respond to any feedback on the draft structure or evaluation section.

Write the introduction, preparation, implementation, and conclusion sections of the dissertation.

Milestones:

- Dissertation completed and sent to DoS and supervisor.

25th Feb – 10th Mar [*Slack*]

I will use this time to finish the dissertation if the first draft is not already finished. If a draft

is finished then I will likely be waiting for feedback and will dedicate this time to revision.

Milestones:

- If the previous milestone was missed then it should now be met.

11th Mar – 24th Mar ***15th Mar**: Last day of Lent term*

Respond to any feedback on the dissertation.

Milestones:

- Dissertation updated following any feedback received.

25th Mar – 21st Apr [*Slack*]

Continue improving the dissertation and responding to feedback.

Milestones:

- Dissertation updated.

22nd Apr – 5th May ***23rd Apr**: First day of Easter term*

Make any final changes and submit the dissertation.

Milestones:

- Dissertation submitted. ***17th May**: Dissertation deadline*

Resources Required

I am intending to use my own laptop for the project (Dell XPS 15, 16GB RAM, i7-7700HQ, running Ubuntu). I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

The basic resources that are essential to my project are the OCaml compiler `ocamlopt` and the SWI-Prolog interpreter `swipl`. These are both installed on the MCS system, so I will be able to complete the project using the MCS machines if my laptop fails.

I will use git for revision control. My git repository will be hosted in GitHub, and this will serve as a cloud backup of my code. I will store my local copy of the git repository inside a Dropbox folder so that my work is also all automatically synced to Dropbox. I will perform weekly backups to a USB stick in case these other backup techniques fail.

References

[1] David H. D. Warren. An abstract Prolog instruction set. *SRI international*, 1983.

[2] Neng-Fa Zhou. A register-free abstract prolog machine with jumbo instructions. In Véronica Dahl and Ilkka Niemelä, editors, *Logic Programming*, pages 455–457, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[3] Andreas Krall. Implementation techniques for prolog. *Institut für Computersprachen Technische Universität Wien*, 1985.

[4] Peter Lodewijk Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, 1990.

[5] OUnit. `hhttp://ounit.forge.ocamlcore.org/`. OCaml unit testing framework.

[6] Jan Wielemaker. Prolog Unit Tests. `http://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/plunit.html%27)`. SWI-Prolog Documentation.

[7] Memory profiling with Spacetime. `https://caml.inria.fr/pub/docs/manual-ocaml/spacetime.html`. OCaml Documentation.

[8] Execution profiling. `http://www.swi-prolog.org/pldoc/man?section=profile`. SWI-Prolog Documentation.

[9] Thomas W. Getzinger. The costs and benefits of abstract interpretation-driven prolog optimization. In *Proceedings of the First International Static Analysis Symposium on Static Analysis*, 1994.

[10] Saumya K. Debray and David S. Warren. Automatic mode inference for logic programs. *The Journal of Logic Programming*, 5(3):207 – 229, 1988.

[11] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for prolog. *Artificial Intelligence*, 23(3):295 – 307, 1984.

[12] Anil Madhavapeddy, Jason Hickey, and Yaron Minsky. *Real World OCaml: Functional Programming for the Masses*. O'Reilly Media, 2013.

[13] Lexer and parser generators (ocamllex, ocamlyacc). `https://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html`. OCaml Documentation.