William Henderson

A Prolog interpreter for the browser

Churchill College
University of Cambridge

May 2025

Submitted as part of the requirements for

Part II of the Computer Science Tripos

Declaration

Proforma

Candidate Number:	**2384E**
Project Title:	**A Prolog interpreter for the browser**
Examination:	**Computer Science Tripos – Part II, 2025**
Word Count:	**11,999**[1]
Code Line Count:	**3,746**[2]
Project Originator:	**The candidate**
Supervisor:	**Prof. Alan Mycroft**

Original Aims of the Project

The original aim of the project was to test the hypothesis that a Prolog implementation targeting WebAssembly, designed from the ground up specifically for the browser environment, may be able to achieve superior performance compared to existing solutions, while preserving tight integration with the browser environment. The project was to explore this hypothesis by building such a Prolog implementation.

Work Completed

Alongside achieving the original aims, I implemented a number of extensions, including a browser-based development environment for the Prolog implementation, a precise garbage collector, and a foreign function interface that allows inline JavaScript code to be executed from Prolog. The project achieved significantly better results than all existing solutions tested in terms of execution time and memory usage. I have made the project publicly available on GitHub[3] and NPM[4] under the MIT licence.

Special Difficulties

None.

[2]Calculated with `texcount`, using `%TC:group table 0 1` and `%TC:group tabular 1 1` to include tables.
[2]Calculated with `cloc`.
[3]`https://github.com/2384E/WebPL`
[4]`https://www.npmjs.com/package/webpl`

Contents

Chapter 1

Introduction

This dissertation explores the use of Prolog in the browser by building a Prolog interpreter, WebPL, that runs in the browser using WebAssembly. This chapter introduces the motivation for the project, briefly discussing the existing landscape of Prolog implementations for the browser, and outlines the aims of the project.

1.1 Motivation

Web pioneer Marc Andreessen claimed in the early days of the Web that "the browser will be the operating system": the browser would become the platform on which modern applications are built [Kosner, 2012]. With increasingly sophisticated applications being delivered through the browser, this vision is proving more and more correct. In the last decade, the development of WebAssembly, a low-level binary instruction format for the Web, has further bridged the gap between browser-based and native applications, enabling web applications to run with near-native performance [Haas et al., 2017].

The Prolog logic programming language is not exempt from this trend. Prolog is used on the Web in a variety of applications, including for browser-based development environments like SWISH [Wielemaker et al., 2015], enforcing dependency constraints in JavaScript packages[1], and as a query language for databases [Wielemaker et al., 2007].

Traditional Prolog web applications typically use a client-server architecture. SWISH is one such application: the SWI-Prolog engine runs on the server, communicating with the browser using Pengines [Lager and Wielemaker, 2014], a library for exchanging Prolog queries and results over HTTP. There are several drawbacks of this model. A dedicated server is required to run the Prolog engine, which may be costly, and latency is introduced by the need for network communication. In addition, care must be taken to mitigate the risks of untrusted code execution. Running the Prolog engine in the browser directly would eliminate these drawbacks; indeed, Flach concludes that "an in-browser solution is probably the future of Prolog on the web" [Flach et al., 2023].

There are several Prolog implementations that run in the browser, which fall into two categories: general-purpose native implementations that have been ported to WebAssembly, which I call *web-ported* implementations, and implementations that have been designed from the ground up specifically for the Web, which I call *web-native* implementations. The former category includes SWI-Prolog [Wielemaker et al., 2012] and Trealla Prolog [Davison, 2020], while a notable example of the latter is Tau Prolog [Riaza, 2024], written in JavaScript. At the time of writing, there are no web-native implementations that use WebAssembly.

[1] `https://v3.yarnpkg.com/features/constraints`

Existing web-ported implementations enjoy WebAssembly's near-native performance but suffer from poor integration with the browser environment and large binary sizes. The resource requirements of applications in the browser differ greatly from those of native applications, particularly in terms of memory, and web-ported implementations are typically heavyweight, not taking this into account.

Web-native implementations, on the other hand, are more tightly integrated with the browser environment, but do not yet have the performance benefits of WebAssembly. Furthermore, JavaScript's memory management is entirely independent of the Prolog engine, so information about the state of the Prolog engine cannot be used to optimise memory allocation and deallocation.

I hypothesise that **a web-native Prolog implementation using WebAssembly, with design guided by the constraints of the browser environment, may be able to achieve superior performance compared to existing implementations, while preserving tight integration with the browser**. The project confirms this hypothesis by building such a Prolog implementation.

1.2 Project Aims and Outcomes

The aims of the project, all of which were achieved, were as follows:

- to **build a web-native pure Prolog implementation** that uses WebAssembly, with a particular focus on performance and browser integration, including
  - a **lexer** and **parser**, and
  - an **interpreter** for the resulting abstract syntax tree,
- to **explore optimisations** to improve performance, including both traditional Prolog optimisations and those that specifically target the browser environment, and
- to **evaluate its performance**, in terms of execution time and memory usage, against existing web-ported and web-native Prolog implementations, exploring which factors contribute to performance differences.

The following extensions were implemented:

- to **extend the implementation** to support more advanced Prolog and CLP features, such as extra-logical predicates, cut, and suspended goals,
- to **build a browser-based development environment** for Prolog, like SWISH, which supports multiple Prolog implementations to facilitate comparison,
- to **implement a garbage collector** to make better use of memory, and
- to **develop a foreign function interface**, extending Prolog syntax to support **inline JavaScript code**, to better integrate with the browser environment.

Chapter 2

Preparation

This chapter introduces Prolog and its execution model (Section 2.1), and how garbage collection techniques can be applied to it (Section 2.2). It also discusses web development with WebAssembly (Section 2.3), before describing the tools (Section 2.4) and development methodology (Section 2.5) used in the implementation of WebPL. Finally, a requirements analysis is performed (Section 2.6) and the starting point of the project stated (Section 2.7).

2.1 Prolog

Prolog is rooted in the fundamental idea that logic can be used as a programming language [Kowalski, 1974]. This section summarises the theory behind Prolog, as well as its execution model and practical implementation details.

2.1.1 Clauses and Terms

Prolog is based on *first-order logic*, specifically *Horn clauses*, which are of the form

$$B \leftarrow A_1 \wedge \cdots \wedge A_n \qquad (n \geq 0)$$

where B is a positive literal (the *head*) and $A_1, \dots, A_n$ are negative literals (the *body*). The key insight of Kowalski's *logic programming* paradigm is that we can interpret Horn clauses as procedures of a simple programming language, written in Prolog syntax as

$$\texttt{B :- A}_1\texttt{, ..., A}_n\texttt{.}$$

A *fact* is a Horn clause with an empty body, and a *goal* is a Horn clause with an empty head.

$$\texttt{B.} \quad \text{(fact)} \qquad\qquad \texttt{?- A}_1\texttt{, ..., A}_n\texttt{.} \quad \text{(goal)}$$

Prolog clauses are built up from *terms*, which may be

- *atoms*, either *constants* (e.g. `foo`, `1209`) or *variables* (e.g. `X`, `Var`), or
- *compound terms*, consisting of a *functor* and a list of *arguments*, themselves terms (e.g. `is_even(X)`, `city(cambridge, united_kingdom)`).

A Prolog program consists of a set of clauses P and a goal clause or *query* Q. The aim of the Prolog execution is to find a *substitution* (an assignment of terms to variables) that makes Q true with respect to P.

2.1.2 Most General Unification

Unification is the process of finding a *substitution* that makes two terms equal. A substitution θ is a set of variable assignments, written $[t_1/x_1, \ldots, t_n/x_n]$, where t_i are terms and x_i distinct variables. A substitution is a *unifier* of two terms t_1 and t_2 if $t_1\theta = t_2\theta$. The *most general unifier* (MGU) θ of two terms is the unifier such that all other unifiers ϕ can be written as the composition of θ and another substitution ψ. Working with MGUs ensures that, if a solution exists, it will not be missed.

Martelli-Montanari Algorithm The *Martelli-Montanari algorithm* is an algorithm for computing the MGU, if it exists, of two terms t_1 and t_2 [Martelli and Montanari, 1982]. It iteratively applies rules to a set of equations, starting with $\{t_1 = t_2\}$, until no more rules can be applied, at which point the remaining set of equations is the unifier.

The rules are shown in Figure 2.1. s and t are terms, X and Y are variables, f and g are function symbols, and S is a set of equations. vars(t) denotes the set of variables occurring in t.

$$\begin{aligned}
\{f(s_1,\ldots,s_n) = f(t_1,\ldots,t_n)\} \cup S &\rightarrow \{s_1 = t_1, \ldots, s_n = t_n\} \cup S \\
\{f(s_1,\ldots,s_n) = g(t_1,\ldots,t_m)\} \cup S &\rightarrow \text{fail} \\
\{X = X\} \cup S &\rightarrow S \\
\{X = t\} \cup S &\rightarrow \{X = t\} \cup S[t/X] \quad \text{if } X \notin \text{vars}(t) \\
\{X = t\} \cup S &\rightarrow \text{fail} \quad \text{if } X \in \text{vars}(t) \wedge X \neq t
\end{aligned}$$

Figure 2.1: The Martelli-Montanari algorithm

The condition $X \notin \text{vars}(t)$ is the *occurs check*, and prevents the creation of infinite terms. In Prolog, the occurs check is often disabled by default for performance reasons, enabling the representation of cyclic terms.

2.1.3 Resolution

Resolution enables the inference of new clauses from existing clauses. In particular, the more restrictive *SLD resolution* suffices for Horn clauses and is the basis of Prolog's execution model.

Given a goal clause `?-` $\mathtt{A}_1$`, ...,` $\mathtt{A}_n$, and a definite clause `C :-` $\mathtt{B}_1$`, ...,` $\mathtt{B}_m$, such that C and some A_i unify with MGU θ, a new goal clause can be derived by applying the SLD resolution rule:

$$\frac{\mathtt{C\ :\!-\ B}_1\mathtt{,\ \ldots,\ B}_m \qquad \mathtt{?\!-\ A}_1\mathtt{,\ \ldots,\ A}_n}{\mathtt{?\!-\ (A}_1\mathtt{,\ \ldots,\ A}_{i-1}\mathtt{,\ B}_1\mathtt{,\ \ldots,\ B}_m\mathtt{,\ A}_{i+1}\mathtt{,\ \ldots,\ A}_n\mathtt{)}\theta}$$

2.1.4 Execution Model

Given a Prolog program consisting of a set of clauses and a query, Prolog attempts to find a substitution that makes the query true by negating the query and searching for a substitution that refutes the negated query, that is, for which the disjunction of the clauses and the negated query is false. SLD resolution implicitly constructs a *search tree* of possible substitutions, and the execution model of Prolog can be understood as a depth-first search of this tree. Consider the following Prolog program.

```
king(united_kingdom, charles).
king(denmark, frederik).
country(united_kingdom).
country(france).
country(denmark).

?- country(C), king(C, K).
```

The execution of this program explores the search tree shown in Figure 2.2.

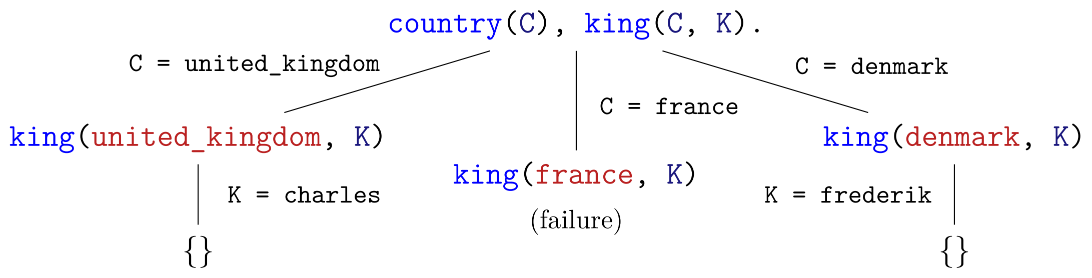


Figure 2.2: Search tree for the Prolog program

The tree is searched depth-first, left-to-right, with clauses selected in program order. Whenever there are multiple candidate clauses to resolve the current goal with, a *choice point* is created, storing the execution state at that point. If a dead end is reached, we *backtrack* to the most recent choice point and try the next possible clause.

The *empty clause* represents a contradiction, so deriving it means that a refutation of the negated query has been reached, and the current substitution is a solution. If all choice points have been exhausted without reaching the empty clause, no refutation can be found, so the query fails.

2.1.5 Implementation

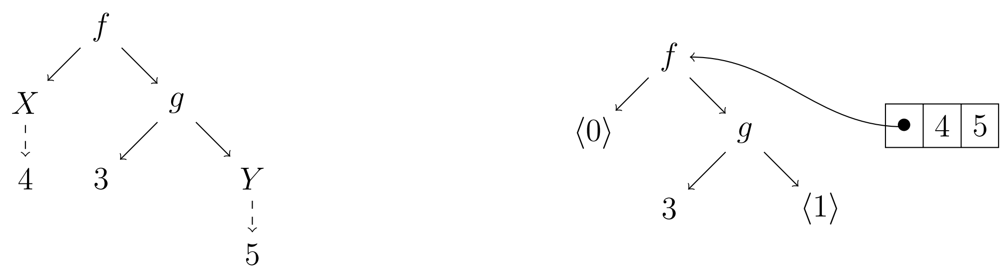
This subsection highlights some key decisions that must be made when building a Prolog implementation.

Unification There are two common ways of implementing the unification subroutine, their signatures depicted in Figure 2.3 in Rust syntax. The first of these does not mutate the terms, instead returning an object representing their MGU, or failing, which I call *substitution-based*. This approach is taken by Tau Prolog [Riaza, 2024]. The second approach directly manipulates the term representations themselves to make them identical, or fails, as in the Warren Abstract Machine (WAM) [Warren, 1983], which I call *mutation-based*.

```rust
fn unify1(a: &Term, b: &Term) -> Option<Substitution>;
fn unify2(a: &mut Term, b: &mut Term) -> Option<()>;
```

Figure 2.3: Two styles of unification

In a *structure-copying* implementation, whenever a clause in the program is resolved with a goal, the terms representing that clause are copied and the variables in the copy bound according to the unifier. A *structure-sharing* implementation recognises that different instances of the same term differ only in their variable bindings, so shares a *prototype* between them, representing the individual instances as *molecules* storing the bindings. Unification is more efficient with the former approach, but the latter can be more memory-efficient [Li, 1998]. These two representations are illustrated in Figure 2.4.



(a) Structure-copying (b) Structure-sharing

Figure 2.4: Representation of the term `f(X, g(Y))` with substitution $\{X \to 4, Y \to 5\}$

Variable Representation In a mutation-based implementation, variables are commonly represented as pointers to the term they are bound to. Unbound variables may be represented by a pointer to themselves, as in the WAM, or by a null pointer. Another approach, taken by Tau Prolog, is to assign an identifier to each variable upon creation, and store the bindings in a separate substitution data structure, which is more amenable to the substitution-based implementation.

Backtracking When backtracking, variables that have been bound since the last choice point must be reset (unbound). This is often achieved by maintaining a stack called the *trail*, which records bindings.

Memory Layout There are several conceptually distinct areas of memory that a Prolog implementation often manages, including

- the *code area*, which stores the clauses or compiled instructions of the program,

- the *local stack*, similar to the stack of a traditional programming language, which stores stack frames containing local terms (those certain not to be used outside the procedure) and return addresses,
- the *global stack* or *heap*, which stores terms that are not local,
- the *control stack*, which stores choice points, and
- the *trail*, which records variable bindings.

Certain Prolog implementations combine some areas. For example, the WAM does not use a separate control stack, preferring to store choice points on the local stack. Tarau's "hitchhiker" virtual machine stores code on the heap rather than in a separate code area, as its encoding of instructions is the same as that of terms [Tarau, 2018]. Li's Logic Virtual Machine (LVM) combines the stack and the heap into a single area to improve locality, which is managed by a garbage collector [Li, 2000].

Last-Call Optimisation (LCO) The last predicate in the body of a clause can re-use the stack frame of its caller and avoid creating a choice point if the clause is *determinate*, that is, if the failure of the last predicate in the body implies the failure of the entire clause. Last-call optimisation enables programs like the following to run in constant stack space.

```
len(L, N) :- len_acc(L, 0, N).
len_acc([], N, N).
len_acc([_|T], A, N) :- A1 is A + 1, len_acc(T, A1, N).
```

2.2 Prolog Garbage Collection

When backtracking to a choice point, any terms that were created since that choice point can be deallocated. This can be achieved by storing pointers to the top of the stack and heap at each choice point, and truncating the stack and heap to these pointers when backtracking. This is called *instant reclaiming* [Bekkers et al., 1992].

However, instant reclaiming is insufficient for *iterative* Prolog programs, that is, those which never backtrack. Consider the contrived program below, which sums the first N natural numbers in constant stack space.

```
sum(N, S) :- sum_acc(N, acc(0), S).
sum_acc(0, acc(N), N).
sum_acc(N, acc(A), S) :-
  N1 is N - 1,
  A1 is A + N,
  sum_acc(N1, acc(A1), S).
```

By wrapping the accumulator in a functor `acc`, we force the implementation to allocate it, at each iteration, on the heap instead of the stack. Since heap memory is only reclaimed when backtracking, and this program never backtracks, the heap grows linearly with N.

To ensure programs like this do not run out of memory, many Prolog implementations use a *garbage collector* to reclaim memory that is no longer reachable.

2.2.1 Mark-and-Sweep

Mark-and-sweep garbage collection operates in two phases. The *root set* contains pointers to live objects, which are recursively followed to mark all reachable objects. Then, the heap is traversed, and unmarked objects are deallocated, often by *compacting* the heap to remove gaps left by deallocated objects.

Warren applied this algorithm to Prolog, using the local stack as the root set [Warren, 1977]. He emphasised, however, that due to the high cost of running the algorithm, it should only be run when necessary (i.e. when the heap is full), and that compile-time classification of variables into locals and globals (to minimise heap usage) is crucial for performance.

2.2.2 Generational Garbage Collection

Generational garbage collection attempts to mitigate the high performance cost of garbage collecting the entire heap by observing that most objects die young (the *generational hypothesis*).

The heap is divided into *generations*, often two: the *young generation*, where new objects are allocated, and the *old generation*, where objects that survive a collection are promoted. Frequent *minor* collections cover the young generation, while infrequent *major* collections cover the entire heap.

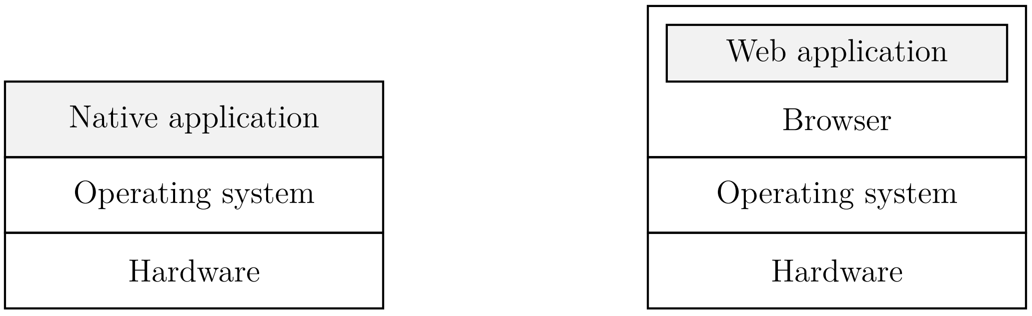
Pointers from the old generation to the young generation complicate this approach. These must be considered part of the root set during a minor collection, so must be tracked by the garbage collector. Fortunately, in Prolog, there is already a mechanism for tracking pointers: the trail. Including the trail in the root set is sufficient to ensure that all reachable objects are marked, and that all pointers are correctly updated, without scanning the entire heap [Bekkers et al., 1992].

2.3 Web Development with WebAssembly

For many years, the platform on which applications would run was the operating system (OS), executing native code but delegating to the OS for privileged functionality such as I/O. I call these applications *native*. However, this requires applications to be built separately for different OSes and architectures, manually installed, and trusted.

For these reasons, increasingly many modern applications are delivered through the browser. I call these applications *browser-based* or *web applications*. The browser, initially only for accessing information, is now the platform on which word processors like Google Docs, videoconferencing applications like Teams, and even editors like VSCode are built. These applications are executed by the browser's JavaScript engine, with the browser providing built-in functions to access lower-level functionality like I/O. In this model, the browser is the only application that needs to be installed and trusted by the user.

Figure 2.5 highlights the difference between native and web applications.



(a) Native application (b) Web application

Figure 2.5: Native and web applications

2.3.1 Why WebAssembly?

JavaScript has been the de facto language for web development since its introduction in 1995. Originally designed to bring interactivity to web pages, and famously built in 10 days, JavaScript has been continuously extended to become the world's most widely used programming language [Wirfs-Brock and Eich, 2020].

As web applications have increased in complexity, the consequences of JavaScript's complicated history have become more apparent. Being a high-level interpreted language, JavaScript is not well-suited for computationally-intensive tasks. Similarly, its poor design as a language, for example its lack of static typing, makes large codebases hard to maintain [Ocariza Jr. et al., 2011, Bierman et al., 2014].

For these reasons, supporting other programming languages on the Web has been a much-discussed topic. Early proprietary attempts, such as Java applets, ActiveX, and Flash, have all since been deprecated due to security concerns[1]. More recently, Emscripten has been used to compile C and C++ code to asm.js, a subset of JavaScript designed to be easily optimised by the JIT compilers of JavaScript engines [Zakai, 2011]. While this approach mitigates the security concerns of earlier technologies, it remains inherently limited by the performance of JavaScript.

WebAssembly was introduced in 2017 as a solution to the problem of running safe, fast, portable, low-level code on the Web [Haas et al., 2017]. It is a binary instruction format for a stack-based virtual machine that is designed to be a compilation target for high-level languages, and able to run in the browser at near-native speed.

[1]Early attempts to support other programming languages on the Web allowed native code to be embedded directly into the web page, which proved difficult to sufficiently "sandbox".

2.3.2 Memory Model

The WebAssembly specification [Rossberg, 2022] is restrictive about how memory is accessed. Two areas of memory that are particularly relevant are the *stack* and the *linear memory*.

Stack A WebAssembly program executes as a *stack machine*: operands of instructions are taken from the top of the stack, and results pushed back onto the stack. The stack also contains labels for control flow instructions, and *activation frames* for function calls, which include local variables. Unlike the stack in a traditional language like C, the WebAssembly stack is not addressable; instead, local variables can only be referenced by their index from the stack pointer, meaning that stack-allocated variables cannot be passed to functions by reference. For example, in the C program below, the compiler must unintuitively allocate the local variable `x` in the linear memory, rather than on the stack, to pass it to `foo`.

```
extern int foo(int*);
int main() {
  int x = 42;
  return foo(&x);
}
```

Linear Memory The main storage of a WebAssembly program is the *linear memory*, or *memory*, a contiguous array of bytes addressed by 32-bit *pointers* (indices into the array). The memory is disjoint from the rest of the WebAssembly program. WebAssembly provides a `memory.grow` instruction to request that the host environment increases the size of the contiguous memory area (an operation like `realloc`), however, there is no way to free memory.

2.3.3 Interface with JavaScript

A key design goal of WebAssembly is to be *language-independent*, thus the WebAssembly specification makes no mention of JavaScript. Instead, it is up to the *host environment* (e.g. the browser) to define and implement the necessary APIs for WebAssembly to interact with the outside world.

Import and Export WebAssembly modules can import and export functions to and from the host environment, allowing JavaScript code to call WebAssembly functions, and vice versa. However, only integers and floats can be passed between the two environments in this way.

Shared Memory To enable more complex data structures to be shared, the browser exposes the WebAssembly module's linear memory to JavaScript as a byte array. This allows JavaScript code to serialise complex data structures into the linear memory, passing a pointer to the WebAssembly module, which can then access the data. WebAssembly functions can return complex data structures in the same way, illustrated in more detail in Appendix A.

2.4 Tools

Rust Rust was chosen as the implementation language for the Prolog interpreter due to its strong support for WebAssembly and its focus on safety and performance. The officially-supported tool `wasm-bindgen` was used to facilitate interaction between Rust and JavaScript, alongside its companion tool `wasm-pack` for building and packaging the compiled WebAssembly module [Crichton, 2014].

Dependencies and Licensing Table 2.1 lists the dependencies used in the implementation of the Prolog interpreter along with their licences, justifying the use of each. Table 2.2 lists the dependencies used in the implementation of the browser-based development environment.

Name	**Licence**	**Justification**
`wasm-bindgen`*	MIT/Apache-2.0	Calls between Rust and JavaScript
`wasm-pack`*	MIT/Apache-2.0	Build and package WebAssembly module
`js-sys`*	MIT/Apache-2.0	Rust bindings to JavaScript APIs
Serde	MIT/Apache-2.0	Serialise and deserialise data
`serde-wasm-bindgen`	MIT	Rust-JavaScript data type conversion
LALRPOP	MIT/Apache-2.0	Generate the parser

Table 2.1: Dependencies of the interpreter. Dependencies marked with a $*$ are officially supported by the Rust WebAssembly Working Group.

Name	**Licence**	**Justification**
React	MIT	JavaScript framework for building UI
Next.js	MIT	Compile React components to static HTML
`react-simple-code-editor`	MIT	Code editor component
Prism	MIT	Syntax highlighting
Iconoir	MIT	Icons

Table 2.2: Dependencies of the browser-based development environment

Version Control Git was used for version control, with the repository hosted on GitHub. I used a trunk-based branching strategy, where each phase of development was carried out on a separate branch, with pull requests used to merge completed features into the `main` branch.

Continuous Integration GitHub Actions, a GitHub service to automatically run code on repository events, was used for continuous integration, compiling and running the test suite on each push to the repository. GitHub's "branch protection" feature was employed to ensure that commits to the `main` branch could only be made through pull requests, and that all checks must pass before merging.

Deployment The browser-based development environment was automatically built and deployed to Cloudflare Pages, a free static website hosting service, on each push to the `main` branch, using GitHub Actions.

2.5 Development Methodology

The project was developed using a *spiral* methodology [Boehm, 1986], with development proceeding in a number of phases, each of which comprising requirements analysis, design, implementation, and testing. Evaluation in each phase involved quantitative comparison to existing Prolog implementations, as described in Chapter 4, alongside qualitative comparison regarding feature completeness, which was then used to inform focus areas for the next phase.

The development phases were guided by the project proposal (Appendix E), beginning with the implementation of a pure Prolog AST interpreter.

The project was tested using unit tests, integration tests, and manual testing. Unit tests were written for each phase to ensure its correctness before proceeding to the next phase. The benchmarking suite, described in Chapter 4, also served as a form of integration testing, as it was run in the browser and required the correct functioning of the system as a whole, with each benchmark program additionally acting as a test case. Manual testing was used to evaluate the browser-based development environment.

2.6 Requirements Analysis

The core requirements as stated in the project proposal (Appendix E) were:

> "The project will be deemed a success if I have written a Prolog interpreter in Rust, compiled it to WebAssembly, and executed it in the browser, as well as having compared its performance with existing solutions, including SWI-Prolog compiled to WASM and Tau Prolog."

In the proposal, a number of extensions were also suggested. Following the background research outlined in this chapter, several of these were selected to be implemented. This research also highlighted the importance of a garbage collector in a practical Prolog implementation, so this was added as a further extension.

- **Cut and Extra-Logical Predicates**: Implementing the cut operator and some extra-logical predicates, which are key components of a full Prolog implementation.
- **Development Environment**: A browser-based development environment to demonstrate and evaluate the interpreter.
- **Garbage Collection**: A garbage collector to improve memory efficiency.
- **JavaScript FFI**: An FFI to improve integration with the browser.

2.7 Starting Point

As detailed in the project proposal (Appendix E), I began the project from scratch, with prior knowledge of Rust. I had little experience with Prolog, having only used it briefly in the Part IB course. No code was written before October 2024.

Chapter 3

Implementation

This chapter gives an overview of the implementation of the project, discussing the implementation and optimisation of the core Prolog interpreter (Section 3.2), its garbage collector (Section 3.3), the JavaScript foreign function interface (Section 3.4), and the web-based development environment (Section 3.5).

3.1 Repository Overview

The repository is structured around four key modules, separated by language and build system. These modules are `core`, the core interpreter written in Rust, `lib`, a higher-level JavaScript library to handle interaction with the WebAssembly interpreter, `web`, the web-based development environment written in TypeScript with React, and `bench`, web-based benchmarking tools written in JavaScript and Python. Figure 3.1 shows the dependencies between these modules.

$$\begin{tikzcd}
\boxed{\texttt{core}} & \boxed{\texttt{lib}} \arrow[l, bend right] & \boxed{\texttt{web}} \arrow[l, bend right] \arrow[r, bend left] & \boxed{\texttt{bench}}
\end{tikzcd}$$

Figure 3.1: Dependencies between modules

Directory	Description	Lines of Code
`core/`	Implementation of the core interpreter in Rust, including the parser, garbage collector, and the JavaScript FFI.	2664 (Rust)
`core/src/wasm/`	WebAssembly interface for the core interpreter and JavaScript FFI.	
`core/src/tests/`	Unit tests for the core interpreter.	
`lib/`	Higher-level JavaScript library for the interpreter.	81 (JavaScript)
`web/`	Web-based development environment to demonstrate and benchmark the interpreter.	836 (TypeScript)
`web/src/prolog/`	TypeScript Prolog interface and wrappers for various Prolog interpreters.	
`bench/`	Web-based benchmarking tools.	165 (JS & Python)
`docs/`	LaTeX source for the dissertation.	N/A
	Total	**3746**

Table 3.1: Directories in the repository

3.2 Core Interpreter

This section describes the implementation of the core Prolog interpreter, WebPL, in Rust.

3.2.1 Memory Layout

WebPL uses the merged heap/stack architecture proposed by Li [Li, 2000] and described in Section 2.1.5. Alongside the advantages identified by Li, notably improved cache performance, this architecture is uniquely suited to the linear memory model of WebAssembly.

In a traditional heap/stack architecture, the heap and stack are separate regions of memory. Running natively, a large virtual address space allows both areas to effectively be unbounded. However, WebAssembly's linear memory is contiguous and must also be carefully managed by the WebAssembly code itself to avoid wasting space. Since the trail stack contains pointers to bound variables, which may appear either on the stack or on the heap, and WebAssembly does not allow pointers to stack-allocated objects (Section 2.3.2), the stack must also be placed in the linear memory. This means that heap/stack implementations must manage two simultaneously-growing stacks in the linear memory (which may only grow upwards), which is difficult to do efficiently.

In a merged heap/stack architecture, all terms are allocated in one contiguous region of memory, avoiding the above dilemma. If the heap grows too large for the linear memory, it suffices to execute the `memory.grow` WebAssembly instruction, with minimal additional work from the WebAssembly module required.

Another key advantage of this architecture is that the heap is stored in chronological order, with older terms at lower addresses. This facilitates the implementation of several garbage collection optimisations, such as variable shunting (Section 3.3.3) and segmented garbage collection (Section 3.3.6).

Of course, other smaller areas are needed, also within the linear memory:

Trail Stack The trail stack is a stack of pointers to variables that have been bound, used for backtracking.

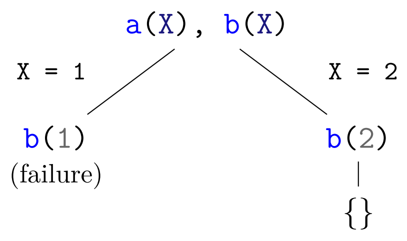
Choice-Point Stack The choice-point stack stores information about where to backtrack to for each choice made. Alongside the next clause to consider, each choice point also contains the sizes of the heap, trail stack and goal stack to facilitate backtracking and instant reclaiming (Section 2.2).

Goal Stack The goal stack contains the remaining goals to prove. When backtracking, goals that have previously been popped might be needed again, so while I refer to it as a stack, the goal stack is actually a graph, with goals represented as pairs (`pred`, `goal`) and the goal stack additionally storing a pointer to the current goal (the "top of the stack").

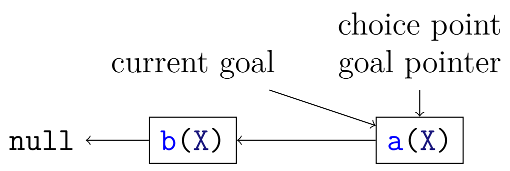
Figure 3.2 illustrates this idea. Before resolving `a(X)` with `a(1)`, a choice point is created, containing a pointer to the current goal (c). After the resolution step, `a(X)` is "popped from the goal stack" by setting the current goal pointer to its predecessor (d). Importantly, however, it remains in the graph, as upon backtracking when `b(1)` fails, the current goal pointer is restored to that stored in the choice point. To prevent the goal stack growing indefinitely, each choice point also contains the number of goals in the stack when it was created, and the goal stack is truncated to this size upon backtracking.

```
a(1).
a(2).
b(2).
?- a(X), b(X).
```

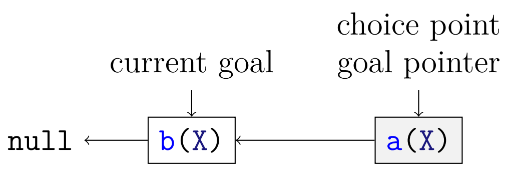
(a) A Prolog program



(b) Corresponding SLD-tree



(c) The goal graph before the first resolution



(d) The goal graph after the first resolution

Figure 3.2: The goal graph

To avoid the need to garbage collect the goal stack, goals are removed from the graph when they are popped if they are not reachable from any choice points (Section 3.2.8).

Code Area There is no separate code area, as the clauses of the program are stored on the heap as ordinary terms, an idea proposed by Tarau for his "hitchhiker" Prolog implementation [Tarau, 2018]. Initially, my implementation used a distinct code area, but profiling revealed that up to 40% of the execution time was spent copying terms onto the heap. Benchmarking confirmed that Tarau's approach reduced mean execution time by around 25%.

Following this approach, copying a clause (`Head :- Body`) onto the heap when a goal unifies with its head consists of a single `memcpy` followed by a linear scan of the copied clause terms to add the appropriate offset to any pointers. To efficiently identify the byte boundaries between terms in the body of the clause, each clause additionally includes a header containing n variables, bound to the n terms in its body (Figure 3.3).

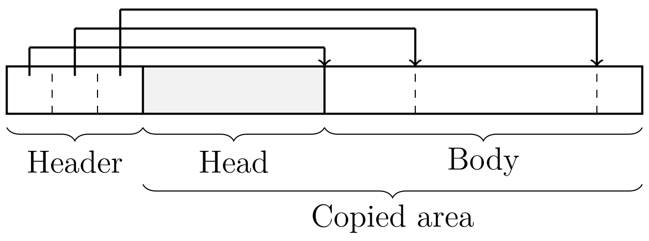


Figure 3.3: Memory layout of a clause on the heap

Lambda Table Each lambda term (representing JavaScript code, Section 3.2.3) contains an index into the *lambda table*. This contains an entry for each lambda term in the program source, which comprises the JavaScript code and the names it uses to refer to its arguments (Section 3.4).

3.2.2 Pointers

A common theme in my implementation, taken from WebAssembly, is that of preferring indexing over pointers. The contiguous nature of the heap allows terms to be referred to by their index, and the same is true for the goal and choice-point stacks.

Rust discourages the use of raw pointers, instead preferring memory-safe *references*. Every object in Rust has exactly one *owner*, with multiple immutable references or a single mutable reference, enforced at compile time by the *borrow checker*. The Prolog heap does not fit nicely into this model: multiple variables may be bound to the same term, all requiring mutable access (e.g. to update the term when unifying). One way to work around this is to use reference counting and `RefCell`, which facilitates runtime borrow checking, but this is slow. Therefore, instead of using references, terms are referred to by their index in the Prolog heap (although this may be informally referred to as a “pointer”). The merged heap/stack architecture makes this straightforward to implement.

3.2.3 Term Representation

Terms are stored on the heap as fixed-size blocks of 16 bytes. These are represented as an enum (Figure 3.5), Rust’s sum type.

Atoms are typed, being strings, integers, or floats. Variables include a pointer to their bound object if bound, or to themselves otherwise, as in the WAM, but additionally a flag to indicate whether they are *shunted* (Section 3.3.3), which is used in garbage collection, and possibly a pointer to a *suspended goal* (Section 3.2.7). This representation enables mutation-based unification. Compound terms, represented by their functor and arity n, are followed in the heap by n variables pointing to each of their arguments (Figure 3.4). This allows their variable-sized arguments to be found in constant time, while keeping their layout contiguous. There is also a heap representation of cut, as goals are stored on the heap. Finally, *lambda terms*, used to call JavaScript code from within Prolog (Section 3.4), are represented by an index into the *lambda table*, along with their arguments like a compound term.

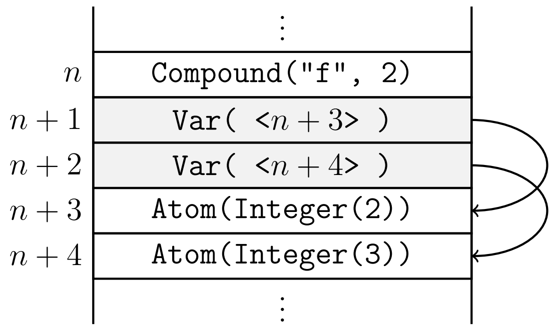


Figure 3.4: Heap representation of `f(2, 3)`

All other kinds of term supported in the syntax are syntactic sugar for terms represented as above, converted during parsing. For example, the arithmetic expression $2 + 3$ becomes the compound term `+(2, 3)`, and the list `[a, b]` becomes the compound term `.(a, .(b, []))`.

```
pub enum HeapTerm {                                          pub enum Atom {
    Atom(Atom),                                                  String(StringId),
    Var(HeapTermPtr, bool, bool, HeapTermPtr),                   Integer(i64),
    Compound(StringId, usize), // followed by args               Float(f64),
    Cut(ChoicePointIdx),                                     }
    Lambda(LambdaId, usize), // followed by args
}
```

Figure 3.5: Term representation in Rust

3.2.4 Parsing

The Prolog program is parsed into its abstract syntax tree using a parser generated by LALRPOP [The LALRPOP Project Developers, 2015] with a custom grammar.

```
Clause: Clause = {
    <h:Term> "." => Clause(h, vec![]),
    <h:Term> ":-" <b:Comma<Term>> "." => Clause(h, b),
}
Comma<T>: Vec<T> = {
    <t:T> => vec![t],
    <mut ts:Comma<T>> "," <t:T> => { ts.push(t); ts },
}
```

Figure 3.6: Extract from the LALRPOP grammar

Figure 3.6 shows the definition of `Clause` and `Comma<T>` in the grammar. LALRPOP supports polymorphic non-terminals, which are used to define a generic non-empty comma-separated list of terms, `Comma<T>`.

3.2.5 Built-in Predicates

To perform useful computation, the interpreter must support some built-in predicates, such as arithmetic evaluation. The full set of built-in predicates is given in Appendix B.

Each predicate implements a Rust *trait*, similar to a C++ interface, called `Builtin` (Figure 3.7). This trait has an associated constant, `ARITY`, the number of arguments the predicate takes. The `eval` function takes a pointer to the first argument and evaluates the predicate, potentially modifying interpreter state through the mutable `Solver` reference, returning a Boolean if the predicate succeeded or failed with no errors, or an error of type `BuiltinError` (e.g. if an argument was insufficiently instantiated).

```
pub trait Builtin<const ARITY: usize> {
    fn eval(solver: &mut Solver, args: HeapTermPtr)
        -> Result<bool, BuiltinError>;
}
```

Figure 3.7: The `Builtin` trait

Many built-in predicates are implemented near-identically, for example the extra-logical type-testing predicates `var/1` and `atom/1`, which only differ in the type they compare against. To avoid code duplication, these predicates are implemented using meta-programming, with a *macro* generating the code for each type (Figure 3.8). It uses Rust's pattern-matching syntax to match the term's heap representation against a pattern `$matcher`.

```
macro_rules! impl_type_check {
    ($name:ident, $matcher:pat) => {
        pub struct $name;
        impl Builtin<1> for $name {
            fn eval(solver: &mut Solver, args: HeapTermPtr)
                -> Result<bool, BuiltinError>
            {
                Ok(matches!(solver.heap.get(args), $matcher))
            }
        }
    };
}
impl_type_check!(IsAtomBuiltin, HeapTerm::Atom(_));
impl_type_check!(IsVarBuiltin, HeapTerm::Var(_, _, _, _));
```

Figure 3.8: Macro for type-testing predicates

3.2.6 Cut

The cut operator, `!/0`, is an extra-logical predicate that commits to the clause in which it appears. In other words, backtracking past a cut causes the predicate it appears in to fail.

When a clause is copied onto the heap, a linear scan updates its variables by the appropriate offset (Section 3.2.1). This also updates any cut goals to include the index of the current choice point, that is, the one before this clause was selected. When the cut becomes the current goal and is executed, the choice-point stack is truncated to the choice point referred to by the cut.

3.2.7 Constraint Logic Programming

Like many Prolog implementations, WebPL includes some features from *constraint logic programming* (CLP). *Attributed variables*, or *suspensions*, allow a term to be attached to a variable, which is executed when the variable is bound [Holzbaur, 1992]. This allows constraints to be attached to variables and checked later when more information is available. Two common uses of attributed variables are in the `freeze/2` and `dif/2` predicates, the former delaying the execution of a goal until a variable is bound to a non-variable term, and the latter ensuring that two variables, when bound, cannot be unified.

I implemented a basic form of attributed variables, where attributes are limited to goals, as described by Carlsson [Carlsson, 1986]. A `delay/2` built-in predicate is used as the basis for this implementation, which takes a term and a goal as arguments. If the term is not yet bound, the goal becomes the term's *suspended goal* in its heap representation, otherwise it is immediately pushed to the goal stack (Figure 3.9). The predicate succeeds in either case. When an attributed variable is bound, its suspended goal is pushed to the goal stack and removed from its heap representation.

```
match &mut solver.heap.data[var] {
    HeapTerm::Var(_, _, has_attr, attr) => {
        *has_attr = true;
        *attr = goal;
    }
    _ => solver.goals.push(goal),
}
```

Figure 3.9: Implementation of `delay/2`

While `delay/2` is sufficient to implement both `freeze/2` and `dif/2`, I implemented `freeze/2` directly as a built-in predicate to improve performance. The implementation of this predicate is identical to that of `delay/2`, except that it attaches the current goal (i.e. `freeze(..., ...)`) to the variable to account for intermediate unifications to variable terms.

3.2.8 Choice-Point Elimination (incl. Last-Call Optimisation)

Last-call optimisation (LCO) is traditionally implemented by reusing the stack frame for the last call of a determinate predicate (Section 2.1.5). This is applicable on the final clause of a predicate, or after a cut.

However, WebPL does not have a call stack, instead storing all terms on the heap and using a goal stack and choice-point stack for control (Section 3.2.1). Therefore, LCO must be implemented in two parts, one on the choice-point stack and one on the goal stack.

For the choice-point stack, I do not create a choice point for the final clause of a predicate, as its failure necessarily implies the failure of the entire predicate, which I call *choice-point elimination*. Static analysis prior to execution is performed to group clauses into predicates to enable this. This has the same effect as traditional LCO: if a clause is determinate, it will not leave any choice points on the stack, as no choice point will be created for either the choice of the clause or the solution of any goals. In the case where a clause cannot be reached because of a cut, this is also sound, as the cut will remove any choice points created by the clause.

Recall the implementation of the goal stack from Section 3.2.1, namely that "popping a goal from the stack" does not remove it from the graph, rather moving the current goal pointer to its predecessor. If goals are only fully removed on backtracking, when the goal stack is truncated, a determinate predicate will use linear space on the goal stack, despite using constant space on the choice-point stack.

However, when no choice points refer to a goal, it is sound to remove it from the graph entirely when it is popped. This is exactly the case in a determinate predicate. The goals of its body are pushed to the goal stack when a goal unifies with its head, and because of its determinacy and choice-point elimination, no choice points are created as these goals are popped and resolved one by one. Furthermore, as they will be allocated at the end of the memory for the goal graph, no fragmentation occurs when they are removed. This is implemented by passing a flag to `Goal::pop`, indicating whether the goal can be safely removed (Figure 3.10).

```
impl Goal {
    pub fn pop(&mut self, determinate: bool) {
        if let Some(ptr) = self.current.take() {
            self.current = self.goals[ptr].prev_ptr();
            if determinate && ptr == self.goals.len() - 1 {
                self.goals.pop();
            }
        }
    }
}
```

Figure 3.10: Implementation of `Goal::pop`

3.2.9 String Interning

Strings are expensive to copy because they must be dynamically allocated in the memory. Comparing two strings of length n for equality is also far more expensive than an arithmetic comparison, requiring loads and $O(n)$ comparisons, as opposed to a single instruction.

For this reason, WebPL avoids using `String` objects wherever possible by using *string interning*. Strings are allocated in a specific area of memory, and references to each string are replaced with its index in that area. A hash table, mapping strings to their indices, prevents duplicates.

A further optimisation initialises the string area with fixed mappings for commonly-used strings, like the names of built-in predicates and arithmetic operators. For example, the string `"+"` is always at index 9, stored as a Rust constant `str::ADD`, so a real string comparison is not necessary, even with runtime-generated strings (due to the uniqueness of the mapping). Figure 3.11 shows how this optimises the arithmetic evaluation code (simplified).

```
match functor {
    str::ADD /*  9 */ => add(&a, &b),
    str::SUB /* 10 */ => sub(&a, &b),
    str::MUL /* 11 */ => mul(&a, &b),
    str::DIV /* 12 */ => div(&a, &b),
    ...
}
```

(a) Arithmetic comparison of functors

```
match lookup_string(functor) {
    "+" => add(&a, &b),
    "-" => sub(&a, &b),
    "*" => mul(&a, &b),
    "/" => div(&a, &b),
    ...
}
```

(b) String comparison of functors (less efficient)

Figure 3.11: Using fixed string indices for faster arithmetic evaluation

3.3 Garbage Collection

Garbage collection is a crucial aspect of the merged heap/stack architecture [Li, 2000]. This section describes the implementation of the garbage collector in WebPL, first discussing its foundation in the mark-and-sweep algorithm (Section 3.3.1), then defining its goal of being "precise" (Section 3.3.2). The optimisations made to achieve this goal are described in Sections 3.3.3 and 3.3.4, with Section 3.3.5 explaining how the implementation was verified to be precise. Finally, Section 3.3.6 describes how generational garbage collection reduced its overhead.

3.3.1 Mark and Sweep

Initially, I implemented a naive mark-and-sweep garbage collector with a compaction phase (Section 2.2.1), based on the algorithm described by Knuth [Knuth, 1997].

The mark phase uses the goal stack, choice-point stack, and named query variables as the root set, marking all terms on the heap that are reachable from these roots. A term is marked by setting its entry in a vector called the *garbage-collection map* (Knuth calls these entries `LINK`s) to `GC_MARKED`. Entries are initialised to `GC_UNMARKED` at the start of each collection.

During the compaction phase, a linear scan of the heap is performed, shuffling marked terms downwards to remove gaps and updating the garbage-collection map with their new locations. The heap is then truncated to the end of the last marked term.

Finally, during the pointer-rewriting phase, the heap is scanned again, and pointers are updated using the garbage-collection map, along with those in the root set.

3.3.2 Precise Garbage Collection

A key aim of this part of the project was to implement a *precise* garbage collector, as defined by Wielemaker and Neumerkel in their paper evaluating garbage collection in various Prolog implementations [Wielemaker and Neumerkel, 2008]:

> "We define a 'precise' garbage collector as a garbage collector that reclaims all data that can no longer be reached considering all possible execution paths from the current state without considering semantics."

They define five programs that must run indefinitely in bounded memory with a precise garbage collector. The final of these concerns if-then-else constructs, which are not implemented in this project, but the remaining four are shown in Figure 3.12.

In order to meet these criteria, a number of Prolog-specific garbage-collection optimisations were made. These are described in the following sections.

```
run :- run(_).
run(X) :- freeze(X, dummy(X)), X = 1, run(T).
dummy(_).
```

(a) Permanent removal of attributes

```
run :- run(_,_).
run(L0, L) :- f(L0, L1), dummy(L1, L).
f([g|X], Y) :- f(X, Y).
dummy(Xs, Xs).
```

(b) And-control (head variables)

```
run :- run(_).
run(X) :- f(X).
run(X) :- X == [].
f([f|X]) :- f(X).
```

(c) Or-control

```
run :- run(_,_).
run(L0, L) :- dummy(L0, L1), f(L1, L2), dummy(L2, L).
f([f|X], Y) :- f(X, Y).
dummy(Xs, Xs).
```

(d) And-control (existential variables)

Figure 3.12: Garbage collection test programs

3.3.3 Variable Shunting

Variable shunting replaces variables that are only seen in their bound state with their binding values [Sahlin and Carlsson, 1991]. Such variables were created and then bound without the creation of any choice points in between. I implemented variable shunting using the "timestamping" algorithm described by Bekkers et al. [Bekkers et al., 1992].

The following program demonstrates the need for variable shunting: `Y` can be shunted to prevent the heap from growing indefinitely.

```
f(X) :- Y = X, f(Y).
?- f(3).
```

Bekkers et al. define the *age* of a variable to be the serial number of the choice point created just after that variable, i.e. the number of choice points on the stack when the variable was created. In the merged heap/stack architecture, due to the chronological layout of the heap, it is equivalent, and simplifies the implementation, to consider the age of a variable to be its index in the heap. The age of a choice point is defined to be the size of the heap upon its creation.

The timestamping algorithm works in two phases, the first during unification and the second during garbage collection, after the mark phase.

1. When unifying a variable with another term, the age of the variable is compared to the age of the latest choice point. If the variable is older, it is pushed to the trail as normal. Otherwise, no choice points exist since its creation, so variable shunting is applicable. The variable is not pushed to the trail and is instead marked as shunted by setting a flag in its heap representation (Figure 3.13).

```
#[inline]
fn unify_var(&mut self, a: HeapTermPtr, b: HeapTermPtr) -> bool {
    if a < self.choice_point_age.0 { self.trail.push(a); }
    else { self.heap.mark_shunted(a); }
    self.heap.unify(a, b);
    true
}
```

Figure 3.13: Variable shunting during unification

2. During garbage collection, a "shunt" phase is added after the mark phase. This phase iterates over the heap, from younger to older terms (high indices to low indices), and for each shunted variable that was marked, performs one of two actions, based on the garbage-collection map (Section 3.3.1) entry of the term it is bound to (Figure 3.14):
   (a) If the term is not yet mapped, the variable is mapped to the term.
   (b) If the term is mapped to an index i, it must be a younger variable that has already been shunted by case (a). Therefore, the variable is also mapped to i.

   The following compaction phase considers all terms that were already mapped by the shunt phase to be dead, and reclaims them. It also performs a final pass over the map to update the entries of all shunted variables to point to their post-compaction locations.

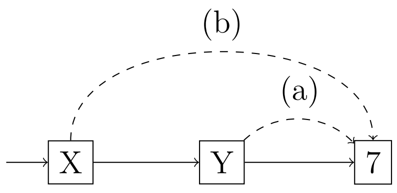


Figure 3.14: Variable shunting during garbage collection. X and Y are shunted variables. Solid lines represent variable bindings. Dashed lines represent the garbage-collection map.

3.3.4 Early Reset

Early reset is concerned with collecting terms that are only accessible from a choice point [Appleby et al., 1988].

Recall the program in Figure 3.12c (or-control). The first clause of `run/1` creates a choice point to try the second clause if the first fails, then iterates through an infinite list. As is, the garbage collector is prevented from reclaiming items already iterated over because they are still reachable from the choice point: in the goal `run(X)`, `X` is bound to the infinitely-growing list by the execution of the first clause. The key insight behind early reset is that `X` will be unbound upon backtracking, before the second clause is executed, so it is safe to reset it "early" if it is only reachable from the choice point.

This is implemented during the mark phase. The root set is ordered such that marking from the choice-point stack is done last, to identify terms only reachable from the choice-point stack. When marking each choice point, any currently unmarked variables that will be reset when backtracking to it (i.e. those that are only reachable from the choice-point stack, if at all) are reset.

This introduces a new problem: variables on the trail stack that have been reset are now garbage themselves. This is resolved by extending the garbage collector to include a *trail stack compaction* phase, which works analogously to the heap compaction phase.

3.3.5 Testing for Precise Garbage Collection

Variable shunting and early reset prove sufficient to meet the criteria for precise garbage collection. To test this, for each of the programs in Figure 3.12, I use the following algorithm:

1. Run the program for a fixed number of iterations to reach a steady state.
2. Perform a garbage collection, then measure the size of the heap.
3. Run the program for a further fixed number of iterations.
4. Perform a garbage collection and measure the size of the heap, passing the test if it is not more than that in step 2.
5. Make one resolution step, then go back to step 4. Repeat this a fixed number of times to get to the same point in the execution as the first measurement, when the test passes.

3.3.6 Generational Garbage Collection

While the naive mark-and-sweep garbage collector augmented with variable shunting and early reset is precise, it is not particularly efficient.

Generational garbage collection avoids scanning the entire heap by dividing it into generations, with the youngest generation being collected most frequently (Section 2.2.2).

Segmented garbage collection applies this concept to Prolog [Appleby et al., 1988]. Recall the concept of *age* from the variable shunting algorithm (Section 3.3.3). This is used to divide the heap into generations, where each generation consists of a choice point and all terms created since it with no choice points in between. When a choice point (and hence a new generation) is created, terms in previous generations cannot become garbage until the choice point is popped upon backtracking. This means that each garbage collection needs only consider terms in generations for which garbage collection has not yet been performed.

This is implemented by keeping track of the age of the oldest choice point for which garbage collection has not yet been performed. Initially, this is set to 0. When a garbage collection is performed, the age is updated to the size of the heap, which would be the age of a new choice point if one were created immediately. When a choice point is popped, the age is updated to the age of the preceding choice point.

3.4 JavaScript FFI

To better integrate the Prolog interpreter with the web environment, I implemented a foreign function interface (FFI) to enable Prolog code to call JavaScript functions. This section details the implementation of this feature.

3.4.1 Augmented Syntax

I augmented the Prolog syntax with a new term type, *lambda terms*, which represent JavaScript code. A lambda term represents a JavaScript function, along with its arguments, which are Prolog terms. Figure 3.15 shows a program that delegates to JavaScript to add two numbers.

```
add(X, Y, Z) :- <{ (X, Y, Z) => unify(Z, X + Y) }>.
```

Figure 3.15: Prolog with embedded JavaScript

To avoid the LALRPOP-generated parser needing to understand JavaScript syntax, it is extended with a separate recursive descent parser for lambda terms. The LALRPOP grammar identifies strings of tokens enclosed in `<{` and `}>` as lambda terms, then invokes the recursive descent parser to parse the function into its code and named arguments (Figure 3.16).

```
LambdaTerm: Term = {
    <js:r"<\{(.|\n)*\}>"> =>? Term::parse_lambda(js)
        .map_err(|error| lalrpop_util::ParseError::User { error }),
}
```

Figure 3.16: Calling the recursive descent parser in LALRPOP

In Figure 3.15, the named arguments are `X`, `Y`, and `Z`, and the JavaScript code is `unify(Z, X + Y)`, which is evaluated when the lambda term is executed.

3.4.2 JavaScript Representation of Prolog Terms

The arguments to such JavaScript functions are Prolog terms, which must be converted to JavaScript objects before the function executes. For ease of JavaScript programming, these are represented using native JavaScript types if possible (e.g. integers as JavaScript numbers). Unbound variables and compound terms are represented using JavaScript objects, the former containing a pointer that is opaque to the JavaScript code (Figure 3.17).

```
                                { functor: "a",
                                  args: [
a(X, b(3, c))    =>                 { variable: (internal pointer) },
                                    { functor: "b", args: [3, "c"] }
                                  ] }
```

Figure 3.17: Prolog terms in JavaScript

3.4.3 Execution of JavaScript

When the current goal is a lambda term, its JavaScript code is executed:

1. Rust converts the arguments to JavaScript objects.
2. Rust calls an external JavaScript function called `eval_js`, passing it the JavaScript code, serialised arguments, and callback functions for internal operations (e.g. unification).
3. `eval_js` converts the code to a JavaScript `Function` object, sets up its environment with the callback functions to enable its interaction with WebPL, then calls it with the arguments (Figure 3.18).

```
try {
    let fn = new Function(...builtin_names, ...arg_names, js)
        .bind(globalThis, ...builtin_values);
    return fn(...arg_values) !== false;
} catch (e) {
    throw e.toString();
}
```

Figure 3.18: Executing the JavaScript function safely

By controlling the environment in which JavaScript code is executed by explicitly providing functions it can call, rather than using `eval`, JavaScript code is prevented from corrupting WebPL's state. The functions available to the JavaScript code are detailed in Appendix C, including `unify`, which calls back into WebPL to unify two terms.

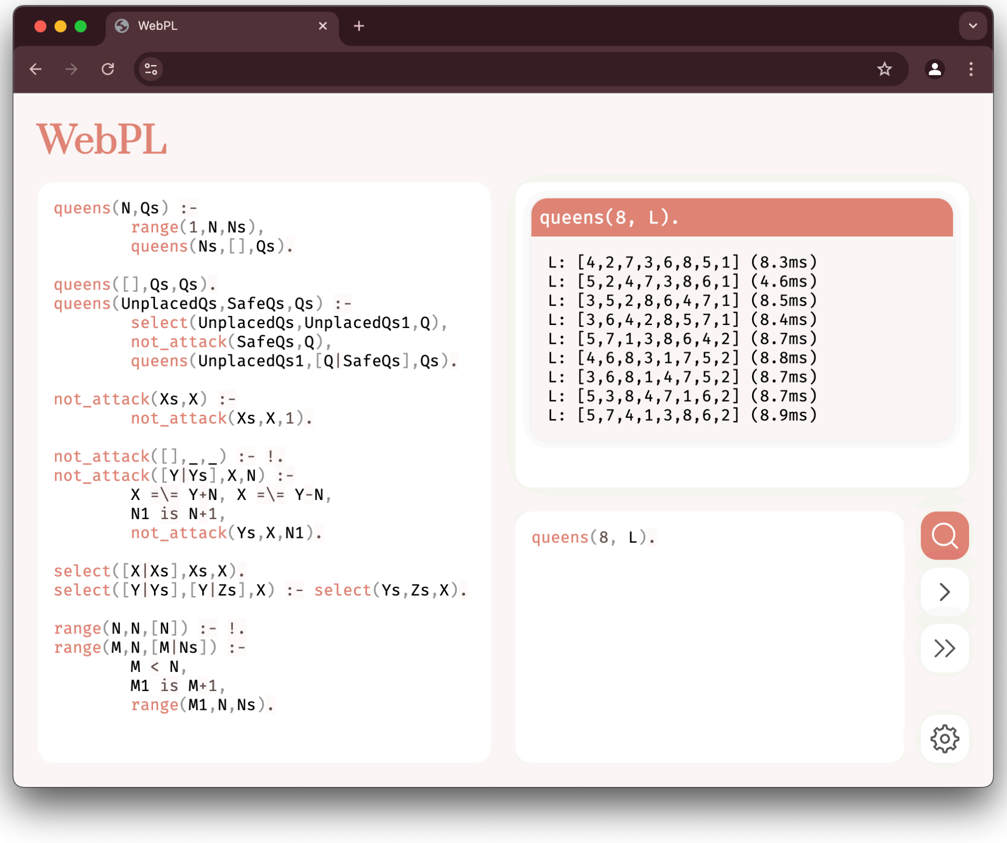
Alongside returning `true` on success, the JavaScript function may return `false` upon logical failure, or throw an exception to indicate an error. In the latter case, the exception is converted to a string before being passed back to Rust.

3.5 Web-Based Development Environment

This section describes the implementation of a web-based development environment for WebPL, inspired by the SWISH Prolog IDE [Wielemaker et al., 2015]. It was written in TypeScript, a strictly-typed superset of JavaScript, to take advantage of its type system and tooling.

3.5.1 User Interface

The user interface (Figure 3.19) was implemented using the React JavaScript framework. It consists of three main components: the program pane, implemented using `react-simple-code-editor`, which provides syntax highlighting; the query pane, where the user can enter a query to run against the program; and the results pane, which displays the results of the query.



```
queens(N,Qs) :-
        range(1,N,Ns),
        queens(Ns,[],Qs).

queens([],Qs,Qs).
queens(UnplacedQs,SafeQs,Qs) :-
        select(UnplacedQs,UnplacedQs1,Q),
        not_attack(SafeQs,Q),
        queens(UnplacedQs1,[Q|SafeQs],Qs).

not_attack(Xs,X) :-
        not_attack(Xs,X,1).

not_attack([],_,_) :- !.
not_attack([Y|Ys],X,N) :-
        X =\= Y+N, X =\= Y-N,
        N1 is N+1,
        not_attack(Ys,X,N1).

select([X|Xs],Xs,X).
select([Y|Ys],[Y|Zs],X) :- select(Ys,Zs,X).

range(N,N,[N]) :- !.
range(M,N,[M|Ns]) :-
        M < N,
        M1 is M+1,
        range(M1,N,Ns).
```

```
queens(8, L).

L: [4,2,7,3,6,8,5,1] (8.3ms)
L: [5,2,4,7,3,8,6,1] (4.6ms)
L: [3,5,2,8,6,4,7,1] (8.5ms)
L: [3,6,4,2,8,5,7,1] (8.4ms)
L: [5,7,1,3,8,6,4,2] (8.7ms)
L: [4,6,8,3,1,7,5,2] (8.8ms)
L: [3,6,8,1,4,7,5,2] (8.7ms)
L: [5,3,8,4,7,1,6,2] (8.7ms)
L: [5,7,4,1,3,8,6,2] (8.9ms)
```

```
queens(8, L).
```

Figure 3.19: The web-based development environment running in Chrome on macOS

3.5.2 Prolog Interface

A key goal of the web-based development environment is to support several Prolog implementations to facilitate comparison. Flach et al. highlight the need for a "standard embedding interface to enable designing interactive web pages using multiple Prolog backends" [Flach et al., 2023], which I aim to provide. Such a TypeScript interface is defined in Figure 3.20.

```typescript
export type Solution = Map<string, string>;
export default abstract class Prolog {
  abstract name: string;
  abstract init(): Promise<void>;
  abstract solve(program: string, query: string): Promise<void>;
  abstract next(): Promise<Solution | undefined>;
  abstract all(): Promise<Solution[]>;
}
```

Figure 3.20: The Prolog interpreter interface

This is an example of the *adapter pattern* [Gamma, 1995], where an adapter is defined for each Prolog implementation, implementing the `Prolog` abstract class using the underlying Prolog system. I implemented adapters for WebPL, SWI-Prolog, Tau Prolog, and Trealla Prolog, allowing them to be used interchangeably in the IDE. This abstraction also forms the basis of the benchmarking system (Section 4.2).

3.5.3 Web Workers

The traditional programming model in the browser is inherently *asynchronous*, as JavaScript is single-threaded and must not block the main thread to maintain a responsive UI. Long-running computations, like solving a Prolog query, do not fit well into this model, as they block for potentially-long periods of time.

Modern browsers provide *Web Workers*, which run JavaScript in a separate thread, communicating with the main thread using an asynchronous message-passing API. This is used by recent Prolog implementations [García-Pradales et al., 2022, Riaza, 2024].

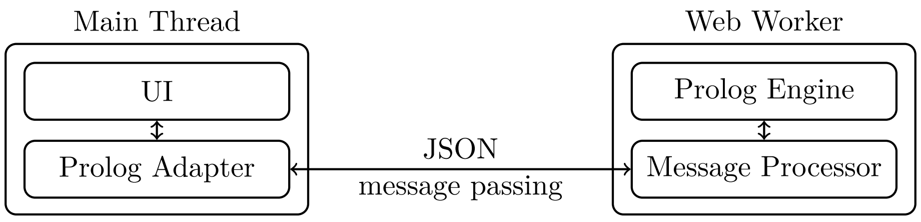


Figure 3.21: Web Worker architecture

To use this approach in WebPL, I added configuration options to the WebPL adapter to run the interpreter in a Web Worker. When the main thread wants to interact with the Prolog interpreter, the adapter sends a JSON message to the Web Worker, which performs the computation and returns the result to the main thread (Figure 3.21).

Since there is no guarantee that messages will be serviced in order, each message is assigned a unique ID which is also present in the response. Before a message is sent, a `Promise` is created, JavaScript's representation of a pending asynchronous operation. This promise is returned to the caller, and its *resolution* and *rejection* functions are stored in a map, indexed by the message ID. When a response is received, the corresponding function is called with the response or error, allowing success or failure to be handled in JavaScript.

Chapter 4

Evaluation

This chapter evaluates the performance of WebPL in terms of execution time (Section 4.2), memory usage (Section 4.3), and binary size (Section 4.4), and compares it to existing Prolog implementations for the Web. Then, factors contributing to the performance differences are explored, and possible improvements to SWI-Prolog are described (Section 4.5). Finally, the use of the Rust programming language is evaluated (Section 4.6).

4.1 Benchmark Programs

A subset of the SWI-Prolog benchmark suite [Wielemaker, 2010], itself derived from a collection of widely-used Prolog benchmarks [Haygood, 1989], was selected to evaluate the performance of WebPL. This consists of 16 benchmarks, excluding the 19 which use features not supported in WebPL.

The benchmarks selected for this evaluation cover a range of tasks, including parsing natural language, symbolic differentiation, list manipulation, arithmetic, and solving puzzles such as the 8-Queens problem. The full benchmark suite is detailed in Appendix D.

4.2 Execution Time

The implementation-agnostic TypeScript interface, developed as part of the browser-based development environment (Section 3.5.2), was further used to build an in-browser benchmarking tool.

Each benchmark was initially run up to 20 times or for 100ms, whichever was shorter, to populate the cache. Then, during the measurement phase, each benchmark was run at least 10 more times, for a maximum of 1000 runs or 5 seconds of benchmarking. The execution time of each run was measured using the browser's `performance.now()` function, which provides high-resolution timestamps. However, high-resolution timestamps are only available in *secure contexts* for security reasons [Sanchez-Rola et al., 2018], so the benchmarking tool was hosted on a local server configured to use HTTPS.

Benchmarks were run in Chrome 134 on a MacBook Pro with an Apple M3 Pro chip and 18GB of RAM, running macOS 15.3 (Sequoia).

4.2.1 Comparison of Prolog Implementations

Table 4.1 shows the median execution time of each benchmark for each implementation tested.

Benchmark	WebPL	WebPL+GC	SWI	Trealla	Tau
chat_parser	17.22	18.51	74.44	26.90	1610.90
crypt	0.64	0.63	3.00	3.62	120.77
derive	0.29	0.28	1.20	0.45	13.25
divide10	0.27	0.24	1.08	0.39	12.99
fib	4.84	7.41	9.72	5.08	3040.73
log10	0.27	0.27	1.08	0.41	13.40
mu	0.34	0.36	1.32	0.62	15.03
nreverse	0.29	0.38	0.65	0.40	17.28
ops8	0.28	0.27	1.10	0.46	13.18
poly_10	5.48	16.98*	19.31	6.45	3481.89
qsort	0.32	0.38	0.84	0.55	18.12
queens_8	6.49	7.30	17.87	8.38	267.43
query	0.77	0.82	2.68	1.26	20.31
tak	15.33	38.56*	28.20	27.96	26691.83
times10	0.27	0.27	1.14	0.57	12.88
zebra	3.16	3.13	7.82	9.76	901.52

Table 4.1: Execution time of benchmarks (milliseconds)

Unexpectedly, but pleasingly, the results show that WebPL is faster than all other Prolog implementations tested in every benchmark without garbage collection, and all but two when garbage collection is enabled (marked with a ∗). Section 4.5 explores why this is, and how some modifications to the SWI-Prolog build process can make its execution time more competitive.

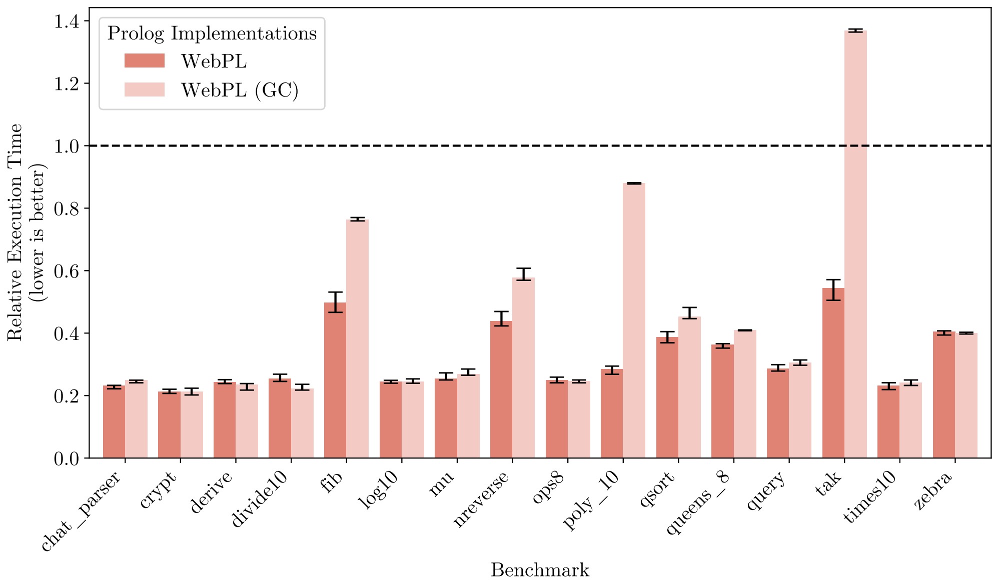


Figure 4.1: Execution time of benchmarks relative to SWI-Prolog

Figure 4.1 shows the execution time of WebPL, relative to SWI-Prolog, based on the same data as Table 4.1, with error bars representing quartiles.

This reveals that for three benchmarks, `fib`, `poly_10`, and `tak`, garbage collection has a major impact on performance, increasing execution time by 53%, 210%, and 152% respectively. These benchmarks are the most memory-intensive, so garbage collection runs more frequently, decreasing performance.

The WebPL garbage collection scheduler invokes the garbage collector whenever heap utilisation exceeds a certain threshold (by default, 90%). If this does not reduce heap utilisation below the threshold, garbage collection is not invoked again until the threshold is met again after the heap has been resized, which occurs when it is completely full. Furthermore, after a garbage collection, there is a timeout before another can begin, to avoid invoking garbage collection too frequently.

This is a more aggressive strategy than that of SWI-Prolog, which does not consider the allocated size of the heap when deciding when to invoke the garbage collector, instead considering the used size of the heap at the last garbage collection[1]. Therefore, SWI-Prolog is less conservative of memory (Section 4.3), and may resize the heap unnecessarily, but avoids the performance penalty of frequent garbage collections.

4.3 Memory Usage

A notable criticism of existing Prolog implementations for the Web is their high memory usage (Section 1.1).

Two approaches were taken in evaluating memory usage. The browser's `performance.memory` API was used to measure the memory usage of a browser tab running the Prolog implementation (Section 4.3.1). However, this includes the size of the WebAssembly binary, located in the memory of the tab. Therefore, the implementation's own memory usage statistics were also used, but this is only available in WebPL and SWI-Prolog (Section 4.3.2).

4.3.1 Web-Page Memory Usage

To evaluate the memory usage of each Prolog implementation for each benchmark, a Python script using the Selenium browser automation library [Software Freedom Conservancy, 2025] was developed. For each implementation and each benchmark, the script starts a new browser instance, loads the implementation and benchmark, runs the benchmark once, and measures the memory usage of the tab using `performance.memory`. As memory used by WebAssembly cannot be freed, the resulting measurement is the peak memory usage during the execution of the benchmark (Table 4.2).

[1]`https://www.swi-prolog.org/pldoc/man?section=gc`

Benchmark	WebPL	WebPL+GC	SWI	Trealla	Tau
chat_parser	6.02	5.58	36.57	26.05	83.05
crypt	4.90	4.90	36.71	25.10	29.72
derive	4.90	4.90	36.95	25.09	5.71
divide10	4.90	4.90	36.45	25.09	5.46
fib	36.77	5.33	36.20	31.78	679.96
log10	4.90	4.90	36.70	25.09	5.46
mu	4.90	4.90	36.45	25.09	5.96
nreverse	4.89	4.89	36.45	25.21	9.46
ops8	4.90	4.90	36.45	25.09	5.46
poly_10	36.78	5.22	36.71	35.84	227.97
qsort	4.89	4.90	36.20	25.22	7.03
queens_8	4.90	4.90	36.46	25.10	69.47
query	4.90	4.90	36.71	25.33	5.72
tak	133.39	36.89	41.01	87.20	3186.35
times10	4.90	4.90	36.45	25.09	5.46
zebra	4.90	4.90	36.45	25.10	104.97

Table 4.2: Memory usage of benchmarks (megabytes)

WebPL shows the lowest memory usage for all benchmarks with garbage collection enabled. However, the variance of memory usage across benchmarks is very small due to the often much larger binary size being included. To better evaluate the memory usage of the Prolog implementations themselves, another approach was taken.

4.3.2 Prolog Heap Usage

WebPL, like SWI-Prolog, provides a built-in `statistics/2` predicate to access statistics about the execution. One such statistic is the allocated size of the Prolog heap. By adding this predicate to the end of the query to be benchmarked, the memory usage of the Prolog implementation itself can be measured. Figure 4.2 shows the allocated heap size of WebPL with garbage collection enabled for each benchmark, relative to that of SWI-Prolog.

For some benchmarks, WebPL uses much less memory than SWI-Prolog. This is because SWI-Prolog pre-allocates a fixed amount of heap memory to avoid the overhead of doing so during execution, while WebPL does not, instead doubling the size of the heap each time it gets full. As a result, WebPL never allocates more than twice the memory it needs.

For other benchmarks, WebPL uses slightly less memory than SWI-Prolog. Only in the case of `tak` does WebPL use more. These are the more memory-intensive benchmarks, possibly indicating that, in the limit, SWI-Prolog is more memory-efficient. The use of Rust for WebPL limits the memory usage optimisations that can be made (Section 4.6).

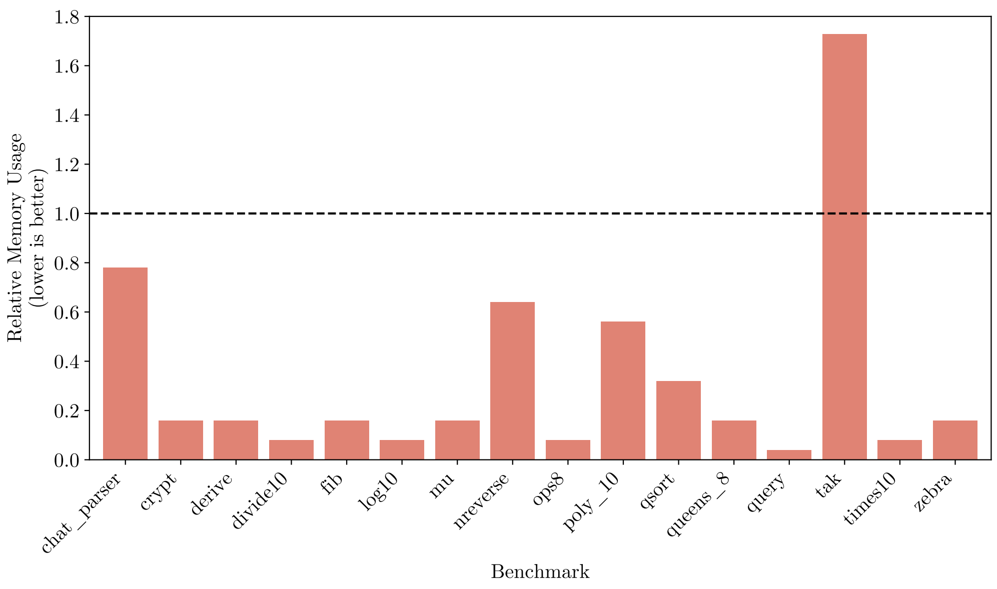


Figure 4.2: WebPL (GC) allocated heap size relative to SWI-Prolog

4.4 Binary Size

Another important consideration for any web application is how much data needs to be downloaded to the client. Unlike native applications, which are downloaded once and run locally, web applications are usually loaded from the server every time a client visits the page. Therefore, the size of the application can have a significant impact on the time it takes to load the page, especially on mobile devices with slower connections. This is a common weakness of WebAssembly applications, as WebAssembly binaries can be large, often due to the inclusion of libraries, such as `libc`, which, unlike in native applications, cannot be dynamically linked.

Table 4.3 shows the size of the WebAssembly binary (for WebPL, SWI-Prolog, and Trealla Prolog) or the JavaScript bundle (for Tau Prolog) for each Prolog implementation.

Implementation	Binary/Bundle Size
WebPL	0.84 MB
SWI-Prolog	7.95 MB
Trealla Prolog	4.48 MB
Tau Prolog	0.57 MB

Table 4.3: Binary/bundle size of Prolog implementations (megabytes)

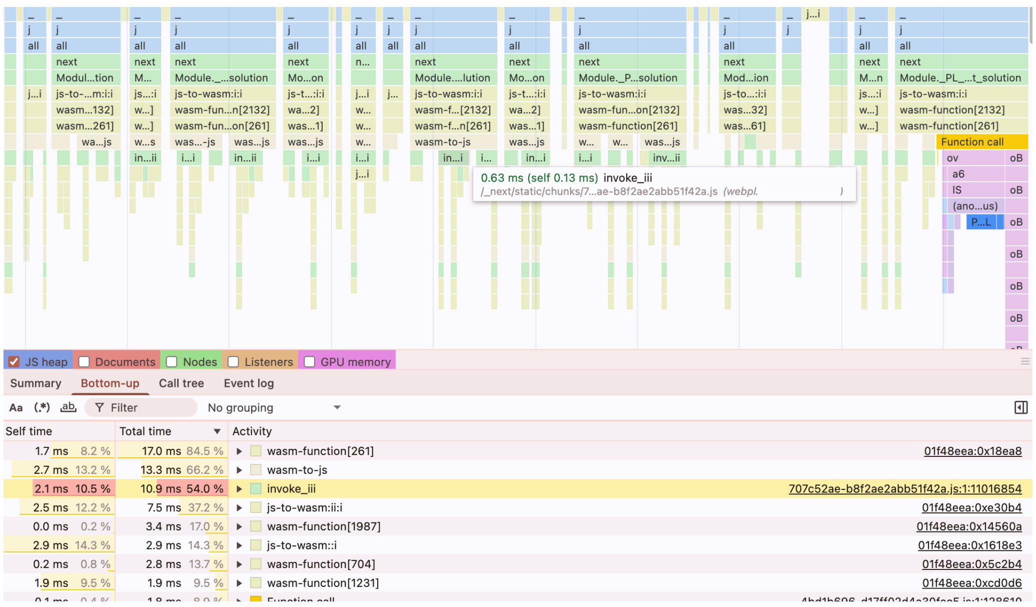
While Tau Prolog, written in JavaScript, is the smallest, WebPL is the only WebAssembly implementation smaller than 1MB, and is only 48% larger than Tau Prolog, significantly less than SWI-Prolog and Trealla Prolog. Section 4.5.3 explores why SWI-Prolog is so large, and how its size can be reduced.

4.5 SWI-Prolog Optimisation

Given the unexpectedly poor performance of industry-standard SWI-Prolog in both execution time and binary size, I explored why this may be, and made some optimisations.

4.5.1 Profiling SWI-Prolog

As identified in Section 4.2.1, WebPL's execution time is significantly faster than that of SWI-Prolog. To explore why, I profiled the execution of the `queens_8` benchmark in SWI-Prolog using Chrome DevTools. Figure 4.3 shows the resulting stack chart.



0.63 ms (self 0.13 ms) invoke_iii
/_next/static/chunks/7...ae-b8f2ae2abb51f42a.js (webpl.

Self time		Total time		Activity	
1.7 ms	8.2 %	17.0 ms	84.5 %	wasm-function[261]	01f48eea:0x18ea8
2.7 ms	13.2 %	13.3 ms	66.2 %	wasm-to-js	
2.1 ms	10.5 %	10.9 ms	54.0 %	invoke_iii	707c52ae-b8f2ae2abb51f42a.js:1:11016854
2.5 ms	12.2 %	7.5 ms	37.2 %	js-to-wasm:ii:i	01f48eea:0xe30b4
0.0 ms	0.2 %	3.4 ms	17.0 %	wasm-function[1987]	01f48eea:0x14560a
2.9 ms	14.3 %	2.9 ms	14.3 %	js-to-wasm::i	01f48eea:0x1618e3
0.2 ms	0.8 %	2.8 ms	13.7 %	wasm-function[704]	01f48eea:0x5c2b4
1.9 ms	9.5 %	1.9 ms	9.5 %	wasm-function[1231]	01f48eea:0xcd0d6

Figure 4.3: Stack chart of SWI-Prolog execution

This revealed that a great deal of time is spent crossing the boundary between WebAssembly and JavaScript code, in particular calling the `invoke_iii` JavaScript function from WebAssembly, which promptly calls back into WebAssembly. This function is generated by Emscripten, the compiler used to compile SWI-Prolog to WebAssembly, and is shown below.

```
function invoke_iii(A, g, I) {
    var C = stackSave();
    try {
        return getWasmTableEntry(A)(g, I)
    } catch (A) {
        if (stackRestore(C),
        A !== A + 0)
            throw A;
        _setThrew(1, 0)
    }
}
```

WebAssembly does not have a built-in exception mechanism, so Emscripten uses JavaScript exceptions instead. `invoke_iii` is used to invoke a WebAssembly function that might raise an exception, and perform the necessary stack manipulation to jump back to WebAssembly code that handles the exception if one is raised. This comes at the performance cost of crossing the JavaScript-WebAssembly boundary not only when raising an exception, but also when calling any function that might do so.

SWI-Prolog is written in C, so does not itself use exceptions. However, it supports Prolog exceptions (extra-logical predicates that can be used to implement more complex control flow), and these are implemented using C `setjmp` and `longjmp` functions. Emscripten uses the same `invoke_iii` mechanism, with its associated cost, to implement these functions.

4.5.2 Experimental WebAssembly Exception Support

While exceptions are not currently supported in WebAssembly, there is a proposal[2] to do so. This has been experimentally implemented in Chrome's V8 JavaScript engine, and can be enabled with the `--enable-experimental-webassembly-features` flag.

To evaluate the potential performance gains of this feature for SWI-Prolog, I modified the SWI-Prolog build process to enable experimental WebAssembly exception support in Emscripten and Node.js, which is used for some SWI-Prolog tests. SWI-Prolog is built using the Docker container system, so I added the necessary flags in various places in the Dockerfile, as well as making further changes to have it build on the Arm architecture.

```
-fno-exceptions -s WASM_EXNREF=1 -s SUPPORT_LONGJMP=wasm
```

The benchmark suite was then re-run with the modified SWI-Prolog build in Chrome with experimental WebAssembly exception support enabled. Figure 4.4 shows the execution time of WebPL and the experimental version of SWI-Prolog, relative to the original version.

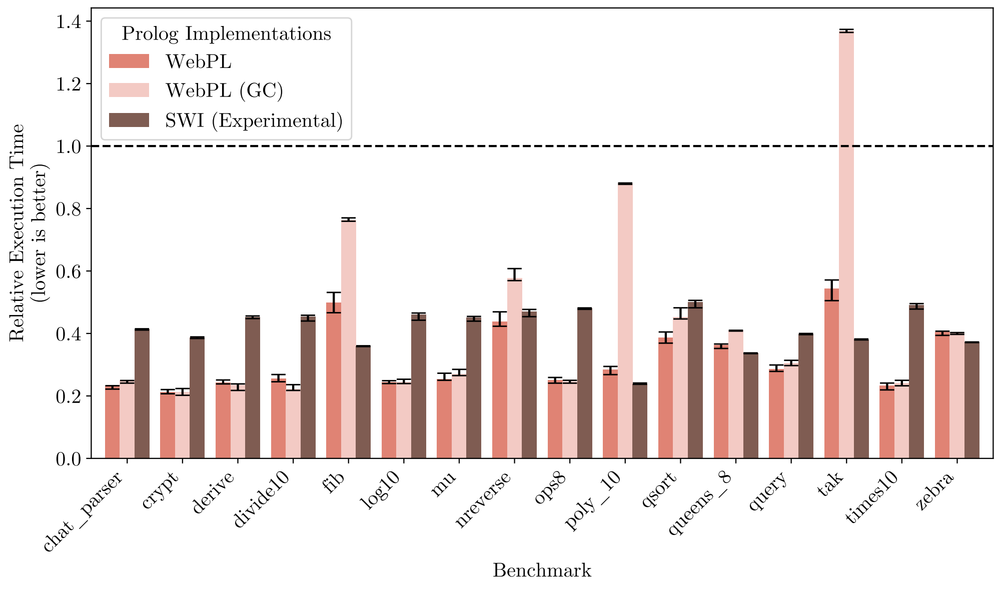


Figure 4.4: Execution time of benchmarks relative to SWI-Prolog

[2]`https://github.com/WebAssembly/exception-handling/blob/main/proposals/exception-handling/Exceptions.md`

This shows that enabling experimental WebAssembly exception support in SWI-Prolog reduces execution time by around 60%, bringing it much closer to, and in some cases surpassing, WebPL.

4.5.3 Binary Size

The size of SWI-Prolog's WebAssembly binary is almost ten times larger than that of WebPL (Section 4.4). It has three distinct parts: the WebAssembly code itself (1.99 MB), libraries (5.80 MB), and JavaScript glue code (0.16 MB). Many of the libraries are not needed in most applications, so I built a "minimal" version of SWI-Prolog as a fairer comparison.

The SWI-Prolog build process provides a `-DSWIPL-PACKAGES=OFF` flag which excludes all libraries except the standard library, reducing the size of the libraries by nearly half to 2.99 MB and the WebAssembly code to 1.23 MB. However, the WebAssembly build depends on the `clib` library to load Prolog code from a URL. Building with just this library enabled results in WebAssembly code of 1.44 MB and libraries of 3.27 MB.

This is still 0.49 MB more than building without `clib`. To avoid this 0.49 MB dependency, I rewrote part of the `wasm.pl` library in the SWI-Prolog source code to use SWI-Prolog's built-in pure-Prolog `url` library instead of `clib`, which is partially written in C.

Figure 4.5 shows the size of each configuration, and compares them to WebPL.

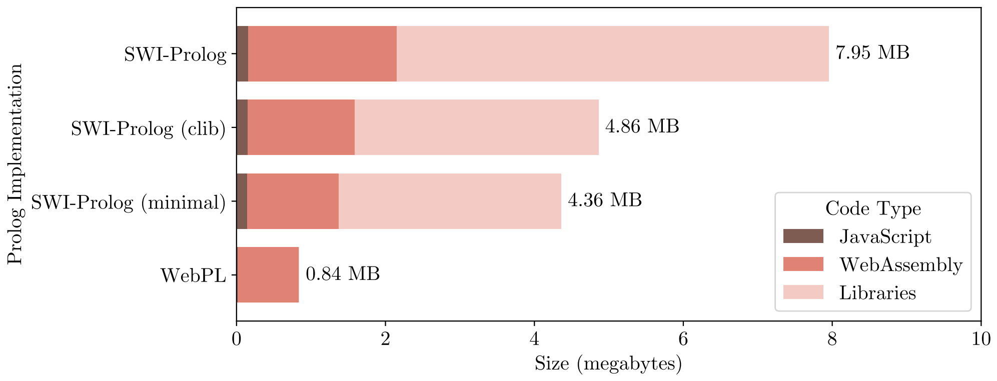


Figure 4.5: Size of WebAssembly binaries for Prolog implementations

Another approach explored to reduce binary size was the `wasm-opt` tool, part of the Binaryen project [Zakai, 2015], configured to optimise aggressively for size using the `-Oz` flag. However, this only reduced the size of the SWI-Prolog binary by 0.38%, and WebPL by 0.62%. This is likely because the compiler is already running code size optimisations by default; indeed, compiling WebPL without optimisations gives a binary size of 14.16 MB, more than 16 times larger than the optimised version.

While WebPL was built from scratch, conscious of its binary size, SWI-Prolog's WebAssembly port was not, and even intentionally avoided some optimisations that would have reduced its binary size in favour of keeping the port closer to the native version[3].

[3] `https://swi-prolog.discourse.group/t/wiki-discussion-swi-prolog-in-the-browser-using-wasm/5651/109`

4.6 Rust Evaluation

WebPL is written in Rust to take advantage of its extensive support for WebAssembly and its performance (Section 2.4). This section explores aspects of Rust that may affect performance.

4.6.1 Unsafe Rust

Rust is known for its memory safety guarantees. Many of these are verified at compile time by the borrow checker, which enforces the ownership and borrowing rules of Rust and prevents common causes of memory errors, such as use-after-free and double-free errors. However, some checks cannot be performed at compile time, such as bounds checking, and are instead performed at runtime, at a performance cost.

Rust provides the `unsafe` keyword to bypass these checks. This is necessary for interfacing with external code that the borrow checker cannot verify, but can also be used cautiously to improve performance.

WebPL represents the heap as an array of fixed-size terms (Section 3.2.1), so indexing into the heap involves a bounds check. By using `unsafe` to bypass this check, the performance of the Prolog interpreter may improve.

Figure 4.6 shows the relative execution time of each benchmark without bounds-checking, compared to the original version of WebPL.

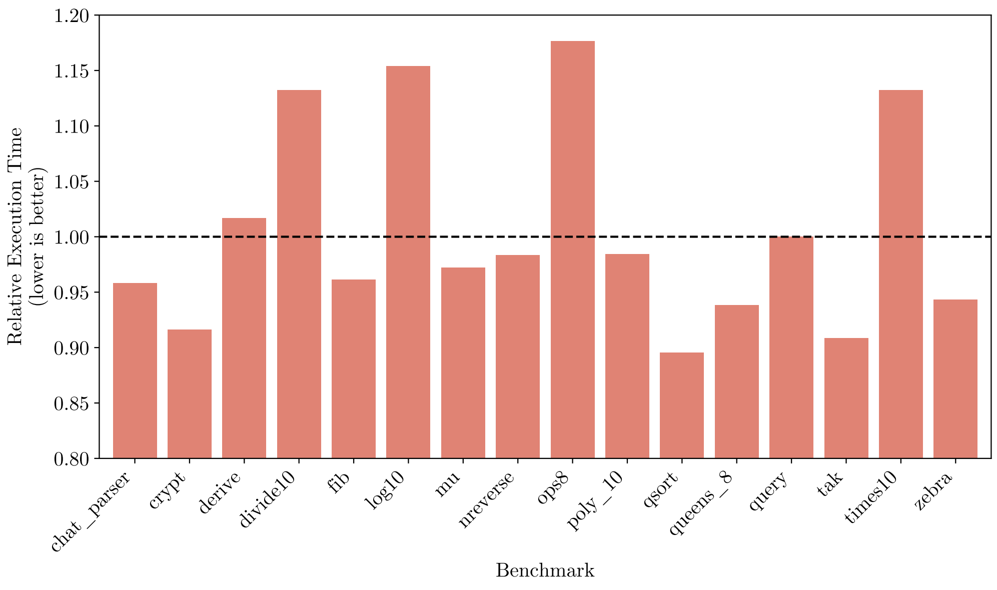


Figure 4.6: Execution time of unsafe WebPL benchmarks relative to safe WebPL

Remarkably, while the more heavyweight benchmarks, like `fib`, `queens_8`, and `tak`, show an improvement of up to 10% lower execution time, the lighter benchmarks, like `divide10`, `ops8`, and `times10`, show a degradation of up to 20% higher execution time. This is likely due to the fact that the overhead of bounds-checking is negligible for lightweight benchmarks, and using unsafe code reduces the compiler's optimisation opportunities.

For this reason, alongside that Rust discourages the use of unsafe code, unsafe code was not used in WebPL.

4.6.2 Memory Layout

Using a Rust `enum` to represent terms leaves their layout in memory up to the compiler, which may use more memory than necessary to represent them. The exact memory layout in WebAssembly can be inspected using experimental Rust compiler flags:

```
$ cargo +nightly rustc --target wasm32-unknown-unknown -- -Zprint-type-sizes
```

Figure 4.7 shows the memory layout of an integer atom term and a variable term in WebAssembly. This reveals significant wasted space: flags `shunted` and `attributed` use a byte each, and the tag indicating the term type uses 4 bytes, even though one would suffice.

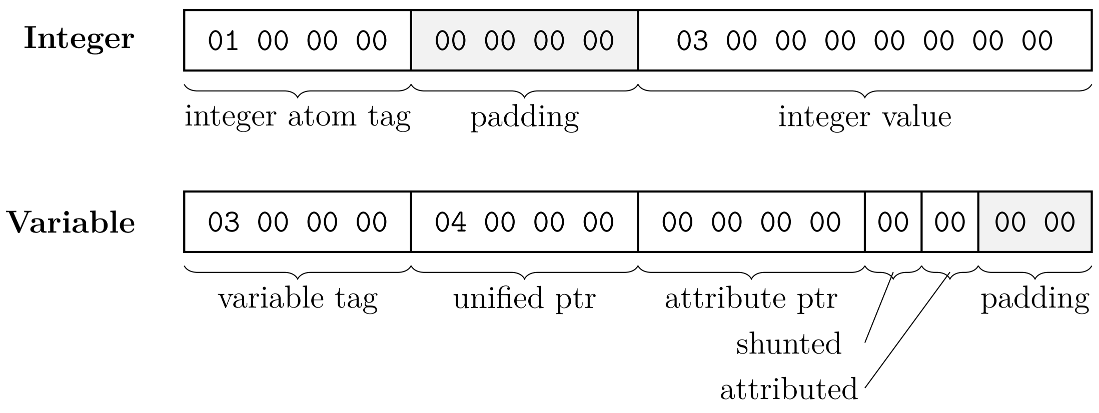


Figure 4.7: Memory layout of Prolog terms in WebAssembly

Padding is used by the compiler to ensure that heap terms are aligned to 8-byte boundaries, which is sensible for performance reasons, and strictly necessary on some architectures. However, this is not the case in WebAssembly, as the WebAssembly specification explicitly states that unaligned accesses must be allowed [Rossberg, 2022], although underlying hardware constraints may mean that they are slower.

Therefore, if a more compact representation of terms were used, consolidating the tag and flags into a single byte and removing padding, the size of each term could be reduced from 16 bytes to 9 bytes, reducing heap usage by 44%. It is possible that maintaining some alignment would still be a worthwhile trade-off, perhaps 4-byte alignment, but this would require more investigation.

Since Rust does not support this level of control over the memory layout without using unions and unsafe code, a pattern that is far from idiomatic, this is not a change that was easily possible to implement in WebPL. However, if the project were to be rewritten in a language like C++, this could be a worthwhile optimisation to explore.

Chapter 5

Conclusions

The project was a success: I built a web-native Prolog implementation using WebAssembly, WebPL, with performance surpassing that of existing Prolog systems, while maintaining tight integration with the browser, thus meeting the success criteria. This demonstrated my original hypothesis (Section 1.1) that such a system could be built.

Furthermore, the extension goals of building a browser-based development environment, implementing a precise garbage collector, and developing a JavaScript foreign function interface were also successfully completed, alongside extending the implementation with extra-logical predicates and CLP features, making WebPL a practical and usable Prolog system for the Web.

Benchmarking showed that WebPL outperforms SWI-Prolog, Trealla Prolog, and Tau Prolog in most cases, even when WebAssembly exception handling is enabled in SWI-Prolog to mitigate its performance issues (Section 4.5). In addition, its memory usage and binary size are both significantly lower than those of other systems, making it suitable for use in resource-constrained environments such as mobile devices.

I have publicly released the project on GitHub[1] and NPM[2] under the MIT licence, and I hope that it may be useful to others.

5.1 Reflections

I learnt a huge amount during this project, especially regarding Prolog, garbage collection, and writing a dissertation. Aspects of the project I anticipated to be straightforward turned out to be challenging, in particular correctly implementing the core Prolog interpreter and the garbage collector. This gave rise to several obscure bugs that required me to gain deep understanding of the nuances of Prolog to remedy.

I particularly enjoyed taking the initial prototype of WebPL and iteratively optimising and redesigning it, guided by benchmarking and profiling, to achieve a final result that was 1000x faster than my first working prototype. I am proud to have built a genuinely usable Prolog system that is fast and lightweight.

Were I to do this project again, I would spend more time on the design of the Prolog interpreter to make it more modular and extensible. I found it very challenging to evaluate alternative design decisions after having made the initial choices, and the ability to configure the system at runtime would have made it easier to experiment with different implementations and allow the user to choose the most appropriate one for their use case.

[1]`https://github.com/2384E/WebPL`
[2]`https://www.npmjs.com/package/webpl`

5.2 Future Work

There are several areas I would have explored further, but lacked time to do so. These include:

- *JIT compilation*: The current implementation of WebPL is an interpreter, with limited pre-execution static analysis to optimise the execution of Prolog code. A just-in-time (JIT) compiler could be implemented to compile Prolog code either to the instruction set of a virtual machine, such as the WAM, or directly to WebAssembly. There are several key questions for such an implementation to address, including how to maintain WebPL's tight integration with the browser and with JavaScript from compiled code.

- *ISO compliance*: Compliance with the ISO Prolog standard [ISO, 1995] is a goal of many Prolog implementations, and one that cements an implementation's credibility. WebPL is not yet fully compliant with the ISO standard, instead implementing a subset of the standard that is sufficient for this dissertation, but full ISO compliance would be a worthy goal for future work.

- *C++ implementation*: Many limitations of WebPL arise from the Rust programming language (Section 4.6). Reimplementing the project in C++ may provide more opportunities for optimisation, and more flexibility and control over the exact behaviour of the system. While C++ does not make the strong memory safety guarantees of Rust, WebAssembly prevents memory safety bugs from causing security vulnerabilities.

Bibliography

[Appleby et al., 1988] Appleby, K., Carlsson, M., Haridi, S., and Sahlin, D. (1988). Garbarge collection for Prolog based on WAM. *Commun. ACM*, 31(6):719–741.

[Bekkers et al., 1992] Bekkers, Y., Ridoux, O., and Ungaro, L. (1992). Dynamic memory management for sequential logic programming languages. In Bekkers, Y. and Cohen, J., editors, *Memory Management*, pages 82–102, Berlin, Heidelberg. Springer.

[Bierman et al., 2014] Bierman, G., Abadi, M., and Torgersen, M. (2014). Understanding TypeScript. In Jones, R., editor, *ECOOP 2014 – Object-Oriented Programming*, pages 257–281, Berlin, Heidelberg. Springer.

[Boehm, 1986] Boehm, B. (1986). A spiral model of software development and enhancement. *SIGSOFT Softw. Eng. Notes*, 11(4):14–24.

[Carlsson, 1986] Carlsson, M. (1986). An implementation of dif and freeze in the WAM. Technical report, SICS, Swedish Institute of Computer Science, Spånga, Sweden.

[Crichton, 2014] Crichton, A. (2014). Wasm-bindgen. https://github.com/rustwasm/wasm-bindgen.

[Davison, 2020] Davison, A. G. (2020). Trealla Prolog. https://github.com/trealla-prolog/trealla.

[Flach et al., 2023] Flach, P., Sokol, K., and Wielemaker, J. (2023). Simply Logical – The First Three Decades. In Warren, D. S., Dahl, V., Eiter, T., Hermenegildo, M. V., Kowalski, R., and Rossi, F., editors, *Prolog: The Next 50 Years*, pages 184–193. Springer Nature Switzerland.

[Gamma, 1995] Gamma, E. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley.

[García-Pradales et al., 2022] García-Pradales, G., Morales, J. F., Hermenegildo, M. V., Arias, J., and Carro, M. (2022). An s(CASP) In-Browser Playground based on Ciao Prolog. In *ICLP Workshops*.

[Haas et al., 2017] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. F. (2017). Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA. Association for Computing Machinery.

[Haygood, 1989] Haygood, R. (1989). A Prolog Benchmark Suite for Aquarius. Technical report, University of California, Berkeley, USA.

[Holzbaur, 1992] Holzbaur, C. (1992). Metastructures vs. attributed variables in the context of extensible unification. In Bruynooghe, M. and Wirsing, M., editors, *Programming Language Implementation and Logic Programming*, pages 260–268, Berlin, Heidelberg. Springer.

[ISO, 1995] ISO (1995). Information technology — Programming languages — Prolog (13211-1:1995).

[Knuth, 1997] Knuth, D. E. (1997). The Art of Computer Programming. Vol. 1 (3rd Edition): Fundamental Algorithms. Addison-Wesley.

[Kosner, 2012] Kosner, A. W. (2012). Always Early: Marc Andreessen's Five Big Ideas That Have Shaped the Internet. *Forbes*.

[Kowalski, 1974] Kowalski, R. (1974). Predicate Logic as Programming Language. volume 74 of *IFIP Congr.*, pages 569–574.

[Lager and Wielemaker, 2014] Lager, T. and Wielemaker, J. (2014). Pengines: Web Logic Programming Made Easy. *Theory and Practice of Logic Programming*, 14(4-5):539–552.

[Li, 1998] Li, X. (1998). A new term representation method for Prolog. *The Journal of Logic Programming*, 34(1):43–57.

[Li, 2000] Li, X. (2000). Efficient memory management in a merged heap/stack prolog machine. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 245–256, Montreal Quebec Canada. ACM.

[Martelli and Montanari, 1982] Martelli, A. and Montanari, U. (1982). An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282.

[Ocariza Jr. et al., 2011] Ocariza Jr., F. S., Pattabiraman, K., and Zorn, B. (2011). JavaScript Errors in the Wild: An Empirical Study. In *2011 IEEE 22nd International Symposium on Software Reliability Engineering*, pages 100–109, Hiroshima, Japan. IEEE.

[Riaza, 2024] Riaza, J. A. (2024). Tau Prolog: A Prolog Interpreter for the Web. *Theory and Practice of Logic Programming*, 24(1):1–21.

[Rossberg, 2022] Rossberg, A. (2022). WebAssembly Core Specification. Technical report, W3C.

[Sahlin and Carlsson, 1991] Sahlin, D. and Carlsson, M. (1991). Variable Shunting for the WAM. Technical Report, European Research Consortium for Informatics and Mathematics at SICS.

[Sanchez-Rola et al., 2018] Sanchez-Rola, I., Santos, I., and Balzarotti, D. (2018). Clock Around the Clock: Time-Based Device Fingerprinting. CCS '18, pages 1502–1514, New York, NY, USA. Association for Computing Machinery.

[Software Freedom Conservancy, 2025] Software Freedom Conservancy (2025). Selenium. https://github.com/SeleniumHQ/selenium.

[Tarau, 2018] Tarau, P. (2018). A Hitchhiker's Guide to Reinventing a Prolog Machine. *OASIcs, Volume 58, ICLP 2017*, 58:10:1–10:16.

[The LALRPOP Project Developers, 2015] The LALRPOP Project Developers (2015). LALRPOP. https://github.com/lalrpop/lalrpop.

[Warren, 1977] Warren, D. H. D. (1977). Implementing Prolog: Compiling Predicate Logic Programs. Technical report, University of Edinburgh.

[Warren, 1983] Warren, D. H. D. (1983). An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, Menlo Park, CA, USA.

[Wielemaker, 2010] Wielemaker, J. (2010). SWI-Prolog benchmark suite.

[Wielemaker et al., 2007] Wielemaker, J., Hildebrand, M., and Ossenbruggen, J. V. (2007). Using Prolog as the Fundament for Applications on the Semantic Web. In *Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services*, ICLP 2007, pages 91–106, Porto, Portugal.

[Wielemaker et al., 2015] Wielemaker, J., Lager, T., and Riguzzi, F. (2015). SWISH: SWI-Prolog for Sharing. In *Proceedings of the International Workshop on User-Oriented Logic Programming*, IULP 2015, pages 99–113, Cork, Ireland.

[Wielemaker and Neumerkel, 2008] Wielemaker, J. and Neumerkel, U. (2008). Precise Garbage Collection in Prolog. In *Proceedings of the 8th International Colloquium on Implementation of Constraint and Logic Programming Systems*, CICLOPS 2008, pages 124–138, Udine, Italy.

[Wielemaker et al., 2012] Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. (2012). SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96.

[Wirfs-Brock and Eich, 2020] Wirfs-Brock, A. and Eich, B. (2020). JavaScript: The first 20 years. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–189.

[Zakai, 2011] Zakai, A. (2011). Emscripten: An LLVM-to-JavaScript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 301–312, New York, NY, USA. Association for Computing Machinery.

[Zakai, 2015] Zakai, A. (2015). Binaryen. https://github.com/WebAssembly/binaryen.

Appendices

Appendix A

JavaScript-WebAssembly Interaction

To illustrate the interaction described in Section 2.3.3, consider the example in Figure A.1 where JavaScript passes a value to WebAssembly through the linear memory, and WebAssembly calls back into JavaScript to output that value. The JavaScript `DataView` API is used to write an integer value to the linear memory, specifying little-endian byte order, and the WebAssembly function `get` loads the value and calls the imported JavaScript function `log` to output it.

```
/* Instantiate the WebAssembly module, providing `console.log` as an external
 * function to call from within WebAssembly. */
WebAssembly.instantiate(fetch("a.wasm"), { env: { log: console.log } })
  .then(({ instance }) => {
    /* Use the built-in DataView API to write a value of a specific size and
     * endianness to the linear memory, in this case writing the integer 1234
     * as a 32-bit little-endian signed integer to address 0. */
    const mem = new DataView(instance.exports.memory.buffer);
    mem.setInt32(0, 1234, /*littleEndian=*/true);

    /* Call the WebAssembly function `get`, passing the address of the value
     * that was just written to the linear memory. */
    instance.exports.get(0);
  });
```

(a) JavaScript code to write to linear memory and call into WebAssembly

```
(module
  ;; Import external function `log` specified when instantiating the module.
  (import "env" "log" (func $log (param i32)))

  ;; Make the linear memory accessible from JavaScript.
  (memory (export "memory") 1)

  (func (export "get") (param $ptr i32)
    local.get $ptr ;; push the pointer onto the stack
    i32.load       ;; load the value from linear memory
    call $log      ;; pass the value to the JavaScript function
  )
)
```

(b) WebAssembly code to read from linear memory and call into JavaScript

Figure A.1: JavaScript-WebAssembly interaction

Appendix B

Built-in Predicates

B.1 Predicates

Supported built-in predicates (Section 3.2.5) are shown in Table B.1.

B.2 Statistics Predicate

The `statistics/2` predicate takes a statistic name as its first argument, unifying the current value of that statistic with the second argument. Available statistics are shown in Table B.2.

Predicate	Description
`is/2`	Unifies the first argument with the result of evaluating the second argument, or fails if they do not unify.
`=/2`	Unifies the first argument with the second argument, or fails if they do not unify.
`>/2`	Succeeds if the first argument is greater than the second argument.
`</2`	Succeeds if the first argument is less than the second argument.
`>=/2`	Succeeds if the first argument is greater than or equal to the second argument.
`=</2`	Succeeds if the first argument is less than or equal to the second argument.
`=\=/2`	Succeeds if the first argument is not equal (arithmetic) to the second argument.
`=:=/2`	Succeeds if the first argument is equal (arithmetic) to the second argument.
`==/2`	Succeeds if the first argument is equal (term) to the second argument.
`delay/2`	Delays the evaluation of the second argument until the first is bound (see Section 3.2.7).
`freeze/2`	Delays the evaluation of the second argument until the first is bound to a non-variable (see Section 3.2.7).
`integer/1`	Succeeds if the argument is an integer.
`float/1`	Succeeds if the argument is a floating point number.
`atom/1`	Succeeds if the argument is an atom.
`var/1`	Succeeds if the argument is a variable.
`nonvar/1`	Succeeds if the argument is not a variable.
`compound/1`	Succeeds if the argument is a compound term.
`number/1`	Succeeds if the argument is a number (integer or float).
`!/0`	Cut operator.
`statistics/2`	Queries runtime statistics, see Table B.2.

Table B.1: Built-in predicates

Statistic	Description
`memory`	Current heap memory usage in bytes.
`allocated`	Current allocated heap size in bytes.
`gc`	Number of garbage collections performed.
`wasm_memory`	Current size of the WebAssembly linear memory in bytes.

Table B.2: Supported statistics

Appendix C

JavaScript FFI Built-In Functions

Function	Description
`unify`	Performs unification. Both terms must be valid JavaScript representations of Prolog terms (Section 3.4.2), which will be converted back to Prolog terms and allocated on the heap if necessary. Variable bindings are checked to ensure they refer to terms that exist.
`fetch`	Makes a synchronous HTTP request to the given URL, returning the response as a string. This is implemented in JavaScript as a wrapper around `XMLHttpRequest`, provided for convenience.
`compound`	Creates a compound term from a functor and some arguments.
`list`	Converts a JavaScript list into its linked-list Prolog representation.

Table C.1: Functions available to JavaScript code

Appendix D

Benchmark Programs

Benchmark	**Description**
chat_parser	Parse natural language
crypt	Cryptomultiplication
derive	Symbolic differentiation
divide10	Symbolic differentiation
fib	Fibonacci sequence
log10	Symbolic differentiation
mu	MU puzzle
nreverse	Naive list reversal
ops8	Symbolic differentiation
poly_10	Polynomial exponentiation
qsort	Quicksort
queens_8	8-Queens problem
query	Query deductive database
tak	Takeuchi function
times10	Symbolic differentiation
zebra	Zebra puzzle

Table D.1: Selected benchmarks from the SWI-Prolog benchmark suite

Appendix E

Project Proposal

The original project proposal is on the next page.

Part II Project Proposal: **A Prolog interpreter for the browser**

2384E
University of Cambridge

October 2024

Introduction

Accessing an efficient Prolog interpreter from the browser is convenient for both education and experimentation. However, at the time of writing, only limited work has been done in this space. The existing solutions fall broadly into three categories: interpreters written in JavaScript, client-server systems where Prolog code is sent to a server for execution, and standard Prolog interpreters like SWI-Prolog [1] compiled to the browser-compatible binary format WebAssembly (WASM).

Interpreters written in JavaScript suffer from performance issues due to the interpreted nature of the JavaScript language, and client-server systems require a server as host. Using a Prolog interpreter compiled to WebAssembly is therefore the most promising solution, but not one that has been explored in great depth.

General-purpose Prolog interpreters, like SWI-Prolog, are relatively heavyweight, and the resource requirements in the browser differ greatly from those when running directly on the operating system, particularly in terms of memory. I hypothesise that a more lightweight interpreter, specifically optimised for the browser use-case and built from first principles, may be able to achieve superior performance. The aim of this project is to test this hypothesis by building such an interpreter in Rust, compiling it to WebAssembly, and evaluating its performance compared to existing solutions.

The project will also involve building a lexer and parser alongside the core interpreter. As an extension, I will build a browser-based IDE to demonstrate the interpreter's functionality, which will also facilitate comparison with other Prolog implementations by allowing the user to choose the underlying interpreter. Unlike existing solutions, my approach will optimise from first principles specifically for the browser use-case.

Starting Point

The project will be implemented in Rust, a language with which I have some experience through a number of personal projects over the last four years, including those targeting WebAssembly. I plan to use Rust's LALRPOP [2] parser generator as part of the project, which I have not used before, but I expect knowledge from the Part IB Compiler Construction course to be relevant.

The browser-based IDE will be implemented using TypeScript with the React framework, with which I have a similar level of experience to Rust, although I have not used WebAssembly with TypeScript to this extent.

I have limited familiarity with Prolog from the Part IB Prolog course.

I have experimented with building a Prolog interpreter in Rust in the past, but do not intend to use any of this code in my project.

Project Substance and Structure

The core part of the project will be the implementation of an interpreter for a pure Prolog AST in Rust, which will be compiled to WebAssembly for use in the browser.

1. *Pure Prolog AST interpreter*: Initially, I will build an interpreter in Rust for a pure Prolog AST, represented as a Rust `enum`.
2. *Lexer and parser*: I will complement the interpreter with a lexer and parser using LALRPOP.

At this stage, I will have a standalone Prolog interpreter with similar functionality to SWI-Prolog. The next step will be compiling it to WebAssembly, which in theory should be as simple as changing the target in the Rust compiler. However, in practice, to interface with JavaScript and the browser for IO and other operations, some additional work will be required, and I intend to use the `wasm-bindgen` [3] Rust library for this.

3. *Run in the browser*: I will compile the interpreter to WebAssembly, making the necessary changes to run it in the browser.
4. *Browser optimisation*: I will measure the performance of the interpreter in the browser, examining the generated WebAssembly code, and make optimisations. This will likely involve exploring the use of different data structures and possibly unification algorithms.

Possible Extensions

A core extension to the project would be the implementation of a browser-based IDE for Prolog to demonstrate the interpreter's functionality, similar to SWI-Prolog's SWISH [4], as well as to facilitate comparison with other Prolog implementations.

Other potential extensions may include:

- Supporting extra-logical predicates such as cut
- Building a JavaScript library to interface with the WASM Prolog interpreter
- Building a Prolog AST to WASM compiler, using the Warren Abstract Machine [5] (or similar), whose generated code can then run in a browser (more difficult)

Success Criteria

The project will be deemed a success if I have written a Prolog interpreter in Rust, compiled it to WebAssembly, and executed it in the browser, as well as having compared its performance with existing solutions, including SWI-Prolog compiled to WASM and Tau Prolog [6] (a JavaScript Prolog interpreter).

Evaluation will, at least, measure performance in terms of execution time and memory usage when running different Prolog programs, including solving the N-queens problem and other problems from the Part IB Prolog notes [7]. The interpreter will include a `time/1` predicate (as in SWI-Prolog) to measure execution time, and the browser's performance tools will be used to measure memory usage. Each program will be run multiple times and the results (discounting the first for cache warm-up) summarised.

Work Packages

- *17th - 30th October (Michaelmas weeks 2 and 3)*: Research existing Prolog interpreters that work in the browser, and explore how they are implemented. Write a draft report about this to form part of the introduction/preparation section of the dissertation.
  Milestone (30th October): complete draft report.
- *31st October - 13th November (Michaelmas weeks 4 and 5)*: Begin to implement the core interpreter for a pure Prolog AST in Rust.
- *14th - 27th November (Michaelmas weeks 6 and 7)*: Continue work on the core interpreter.
  Milestone (27th November): core interpreter complete.
- *28th November - 12th December (Michaelmas week 8 and Christmas vacation)*: Extend the interpreter to include a lexer and parser using LALRPOP.
  Milestone (12th December): lexer and parser complete.
- *13th December - 25th December (Christmas vacation)*: Revise for exams and catch up on any work that has fallen behind.
- *26th December - 8th January (Christmas vacation)*: Evaluate the performance of the interpreter in the browser, make optimisations, and compare to existing solutions.
  Milestone (8th January): meet success criterion.
- *9th January - 22nd January (Christmas vacation)*: If on track, begin work on the browser-based IDE extension, otherwise catch up on any work that has fallen behind.
- *23rd January - 5th February (Lent weeks 1 and 2)*: Write the progress report, due on 7th February.
  Milestone (5th February): complete progress report.
- *6th February - 19th February (Lent weeks 3 and 4)*: Catch up on any work that has fallen behind or work on an extension.
- *20th February - 5th March (Lent weeks 5 and 6)*: Write the introduction and preparation sections of the dissertation and get feedback from supervisor.
  Milestone (5th March): complete draft introduction/preparation sections.
- *6th March - 19th March (Lent weeks 7 and 8)*: Write the implementation section of the dissertation and get feedback from my supervisor.
  Milestone (19th March): complete draft implementation section.
- *20th March - 2nd April (Easter vacation)*: Write the evaluation and conclusion sections of the dissertation, get feedback from my supervisor, and spend time away from the project revising for exams.
  Milestone (2nd April): complete draft dissertation.
- *3rd April - 16th April (Easter vacation)*: Revise for exams and respond to feedback.
- *17th April - 30th April (Easter vacation)*: Revise for exams and respond to feedback.
- *1st May - 14th May (Easter weeks 1 and 2)*: Respond to feedback and make final changes to the dissertation if necessary, due on 16th May.
  Milestone (14th May): dissertation submitted.

Resources Declaration

I will be using my own laptop (2023 MacBook Pro, Apple M3 Pro (11-core CPU, 14-core GPU), 18GB RAM, MacOS Sequoia). If this fails, I will use my old laptop (2018 Surface Pro 6, Intel i5-8250U, 8GB RAM, Windows 11) until I am able to replace it. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. I will make frequent commits to a private GitHub repository alongside regular backups to Google Drive.

References

[1] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[2] The LALRPOP Project Developers. LALRPOP. https://lalrpop.github.io/lalrpop/, 2015. Accessed: 2024-10-09.

[3] Alex Crichton. wasm-bindgen. https://rustwasm.github.io/wasm-bindgen/, 2014. Accessed: 2024-10-09.

[4] Jan Wielemaker, Torbjörn Lager, and Fabrizio Riguzzi. SWISH: SWI-Prolog for sharing. *arXiv preprint arXiv:1511.00915*, 2015.

[5] David H. D. Warren. An Abstract Prolog Instruction Set, 1983.

[6] Jose A Riaza. Tau Prolog: A Prolog Interpreter for the Web. *Theory and Practice of Logic Programming*, 24(1):1–21, 2024.

[7] Ian Lewis. Prolog. https://www.cl.cam.ac.uk/teaching/2324/Prolog, 2024. Accessed: 2024-10-09.