

**MPIR, a procedural Programming Language promoting maintainability through Refinement Types & Tag Associated Documentation.**

**Toby Benjamin Clark**

**psytc9@nottingham.ac.uk**

**Word Count: 1590**

Page Count: 6

Number of Revisions: 24

**Project Proposal Document**  
Friday, 29 September 2023

*Supervised by*[**Dr. Ulrik Buchholtz**](https://ulrikbuchholtz.dk/)

**lizub@nottingham.ac.uk**

# Table of Contents

[1 Proposal Introduction 1](#_Toc146891100)

[1.1 Documentation & Maintainability Features 1](#_Toc146891101)

[1.2 Refinement Type System & SMT Solving 1](#_Toc146891102)

[1.3 Maintainability, Futureproofing & Transpilation 1](#_Toc146891103)

[2 Background Information & Motivation 2](#_Toc146891104)

[2.1 Motivation for adopting a Refinement Type System 2](#_Toc146891105)

[2.2 Modern Language Syntax from a HCI Perspective 2](#_Toc146891106)

[2.3 Personal Motivation & Interest 2](#_Toc146891107)

[3 Objectives & Goals of MPIR 3](#_Toc146891108)

[3.1 Parsing & Language Syntax Objectives 3](#_Toc146891109)

[3.2 Semantic Analysis & Compilation Goals 3](#_Toc146891110)

[3.3 Maintainability, Usability & Flag Support Objectives 3](#_Toc146891111)

[3.4 Target Language Generation Objectives 3](#_Toc146891112)

[4 Work Plan & Gannt Chart 4](#_Toc146891113)

[4.1 Gannt Chart 4](#_Toc146891114)

[4.2 Work Plan & Methodology 4](#_Toc146891115)

[4.3 LSEPI Consideration 4](#_Toc146891116)

[5 Bibliography/Works Cited 5](#_Toc146891117)

# Proposal Introduction

Programming Languages, and the way programmers have interacted with them, have evolved rapidly over the past century. Syntactic expressions and statements have **grown more abstract** (Carbonnelle, 2023), as **processors have become faster** (DB, 2023), and more widely available.

However, the path this series of developments has taken may have meandered from what I envision a Programming Language could be, focusing heavily on **security and optimization**, rather than **maintainability and readability**. Many modern languages, such as [Rust](https://www.rust-lang.org/), [Go](https://go.dev/), and [Dart](https://dart.dev/) are optimized for **code safety**, and this is engrained in **not just the way computer scientists use them**, but the **very syntax of the language**.

## Documentation & Maintainability Features

Aside commenting[[1]](#footnote-1), most languages lack **documentation features** built into the language, and oftentimes rely on associated tools, such as [Doxygen](https://www.doxygen.nl/) and [JavaDoc](https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html) to generate code documentation. MPIR intends to create **compiler enforced associations** between variables, their usage, and meaning within the codebase. This aims to alter the way users think about **not just data, but variables themselves**. Having to explain code as you are writing it, can help users think **deeper about their program**, and therefore **produce better code**.

This can be thought of as an **expansion to the current, widely used variable system**: where a variable has a type, value and scope declared through the language syntax, and additional comments about the variable are written alongside it. MPIR aims to **absorb variable context information into the variable itself, accessible through a tag system, allowing additional, custom tags that can be enforced at compile-time and automatically generated tags, such as *author*, *date-modified.***

## Refinement Type System & SMT Solving

In addition to this, MPIR also aims to promote the creation and usage of **Refinement Types**[[2]](#footnote-2), enabling an environment where **programmers can concisely outline the usage and boundaries of their data**, which will be very useful not just for **debugging**, but for clearly **defining the meaning of the code** to allow for **transparency with future maintainers**.

An additional benefit of a **Refinement Type System** is the ability to **statically verify function calls** at **compile-time**, using an **SMT Solver[[3]](#footnote-3)** such as [Microsoft Z3](https://www.microsoft.com/en-us/research/project/z3-3/).This allows for more precise **dead-code elimination** and therefore can lead to faster runtime processing, alongside **notifying the programmer**, in case they have made a mistake.

## Maintainability, Futureproofing & Transpilation

Transpilation is the process of **compiling a source language to a target language**, such as converting a [Python](https://www.python.org/) program to a [Ruby](https://www.ruby-lang.org/en/) program. Since the languages we use are constantly changing, MPIR aims to future-proof code by offering multiple target languages, allowing 1 master algorithm to be deployed to a variety of language.

# Background Information & Motivation

For many programmers, when first starting a project, we ask the question "What programming language should I choose?". This oftentimes leads into the **evaluation of the needs of the program**, and the **selection of the associated language**.

|  |  |  |
| --- | --- | --- |
| **Optimized, High-Speed Languages** | [C](https://www.iso.org/standard/74528.html)**,** [C++](https://en.wikipedia.org/wiki/C%2B%2B)**,** [Go](https://go.dev/)**, …** | (Miller, 2022) |
| **Safe Languages** | [Rust](https://www.rust-lang.org/)**,** [Ruby](https://www.ruby-lang.org/en/)**,** [C#](https://learn.microsoft.com/en-us/dotnet/csharp/)**, ...** | (Gaynor, 2019) |
| **Multi-platform Languages** | [Kotlin](https://kotlinlang.org/)**,** [Dart](https://dart.dev/)**, ...** | (Taylor, n.d.) |
| **Easy-to-learn Languages** | [Python](https://www.python.org/)**,** [Java](https://www.java.com/en/)**, ...** | (III, 2023) |
| **Dependently Typed Languages** | [Agda](https://wiki.portal.chalmers.se/agda/pmwiki.php)**,** [Idris](https://www.idris-lang.org/)**,** [Coq](https://coq.inria.fr/)**, …** | (Slant, 2021) |
| **Easy-to-maintain Languages** | - |  |

With 90% of software life-cost relating to ongoing maintenance plans (Sayed Mehdi Hejazi Dehaghani, 2012), features centred around this haven’t developed much past **basic commenting**. This is one of the driving factors behind the ideas surrounding MPIR, to design and evaluate a language centred around **maintainability** and **source-code longevity**.

## Motivation for adopting a Refinement Type System

Outside of [Liquid Haskell](https://ucsd-progsys.github.io/liquidhaskell/), Refinement Types are a rarely seen mechanic within general-use programming languages, the applications allow for complete, situation-specific type declarations that can be used to verify data. As seen in the example below, providing the constraints of a password as a type, rather than a series of selection statements, allows for a **concise** definition that is **easier to update and maintain**.

**type** password = { *s*: **String** | **L**(*s*) > 8 ^ **S**(*s*) ≥ 1 }[[4]](#footnote-4)

where:

1. **L**(*s*: **String**) → **Int**, returns the length of string *s*.
2. **S**(*s*: **String**) → **Int**, returns the total number of symbols within string *s*.

## Modern Language Syntax from a HCI Perspective

Oftentimes, Programming Language Syntax is based on that of other programming languages, such as C-Style, or Python-Style syntax (Tourville, n.d.). MPIR aims to develop its syntax **adopting natural language techniques** and evaluating through **HCI-style benchmarking methodologies**. In addition to this, syntax-decisions will be influenced by what users find prefer in existing languages (SIG Meeting at CHI'2016, 2016). Through these processes, MPIR aims to provide and test a syntax that is:

1. Transparent & Perceptible
2. Clear & Understood
3. Logically Complete and Typed

## Personal Motivation & Interest

My interest in Computer Science & Programming stems from a fixation on **structure**, and having developed small parsers in the past, I have been **keen to formulate an entire language for a couple of years now**, arguably since encountering the (somewhat horrific) if-then syntax prevalent in [LUA](https://www.lua.org/). Coupling this, with a keenness for documentation, good quality code and program-completeness has led to the idea of **MPIR**.

# Objectives & Goals of MPIR

The development of MPIR will prioritise **working**, **tested** **features** and **well-written documentation** to stay aligned to the very ideologies it promotes. Following a work-flow similar to Boehm’s Spiral (Boehm, 1986), will allow for the **development of high-quality features**, with ample opportunity for the refinement of these features at a later date.

## Parsing & Language Syntax Objectives

The following objectives are relevant to the parser, lexer, and general syntax-development of MPIR. The language-design of MPIR should be heavily influenced by HCI-design practices.

1. Develop a **complete**, **comprehensive** **syntax** for MPIR, expressed as a Context-Free-Grammar.
2. Configure the MPIR Context-Free-Grammar to satisfy LL(1).[[5]](#footnote-5)
3. Develop Unit Testing for the MPIR Lexer & Parser, using sample files.
4. Develop & Test MPIR Lexer & Parser.

## Semantic Analysis & Compilation Goals

This section comprises of goals relating to semantic-analysis and general compilation processes. MPIR aims to be heavily typed, with full variable information to be provided, promoting deeper thought about a user’s code.

1. Develop & Test Type Checking & Scope Resolution
2. Develop & Test a Refinement-Type System
3. Develop & Test SMT Solving of Refinement Type Expressions using [Microsoft Z3](https://www.microsoft.com/en-us/research/project/z3-3/)
4. Develop & Test Comment Generation for Target Language for Refinement Types.

## Maintainability, Usability & Flag Support Objectives

With Maintainability & Usability as decisive factors influencing the design of MPIR, the following objectives relate to furthering the presence of these values within the language.

1. Inbuilt & User Defined Associations
2. Allow Compiler Flags
   1. Enable/Disable Semantic Analysis Processes
   2. Enable/Disable Comment Generation
   3. Enforce Variable Association Flags
3. Provide GUI Interface for declaring Compiler Flags
4. Provide CLI Interface for declaring Compiler Flags
5. Provide Rudimentary IDE for MPIR

## Target Language Generation Objectives

This section describes the objectives relating to target language generation; however this may be subject to change.

1. Provide Target Language for at-least one of the following Languages:
   1. [Python 3.10](https://www.python.org/downloads/release/python-3100/)
   2. [C](https://www.iso.org/standard/74528.html)/[C++](https://isocpp.org/)
   3. [LLVM IR](https://llvm.org/docs/LangRef.html)

# Work Plan & Gannt Chart

This section provides an overview of my annual work plan for my computer science dissertation. It outlines my methodology and includes contingency plans to address any potential difficulties or challenges.

## Gannt Chart



## Work Plan & Methodology

MPIR will be developed using a blend of Boehm’s Spiral & Waterfall Methodologies. Starting with development of the Syntax, CFG & Parser before working on target language generation & SMT Solving. In-case of contingency or disruption to this work plan, I’ve allocated extra time around each task, to either start earlier or finish later.

## LSEPI Consideration

To conform with both University Policy and Legal Guidance, Law, Social, Ethical and Professional consideration will be evaluated twice during this project, as shown by LSEPI on the Gannt Chart.

# Bibliography/Works Cited

Boehm, B., 1986. *ACM Digital Library.* [Online]   
Available at: https://dl.acm.org/doi/10.1145/12944.12948  
[Accessed 27 September 2023].

Carbonnelle, P., 2023. *PYPL.* [Online]   
Available at: https://pypl.github.io/PYPL.html  
[Accessed 26 September 2023].

DB, S. C., 2023. *CPU DB.* [Online]   
Available at: http://cpudb.stanford.edu/  
[Accessed 26 September 2023].

Gaynor, A., 2019. *Prossimo.* [Online]   
Available at: https://www.memorysafety.org/docs/memory-safety/#:~:text=Since%20memory%20safety%20bugs%20are,Swift%2C%20Python%2C%20and%20JavaScript.  
[Accessed 27 September 2023].

III, N. D., 2023. *BestColleges.* [Online]   
Available at: https://www.bestcolleges.com/bootcamps/guides/6-easiest-programming-languages-to-learn/  
[Accessed 27 September 2023].

Miller, S., 2022. *q.* [Online]   
Available at: https://www.codecademy.com/resources/blog/what-is-the-fastest-programming-language/  
[Accessed 27 September 2023].

Sayed Mehdi Hejazi Dehaghani, N. H., 2012. *National Library of Medicine.* [Online]   
Available at: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3610582/#:~:text=Software%20maintenance%20costs%20are%20rising,shown%20in%20the%20Figure%201.  
[Accessed 27 September 2023].

SIG Meeting at CHI'2016, 2016. *Usability of Programming Languages.* [Online]   
Available at: https://homepages.inf.ed.ac.uk/wadler/papers/chi-sig-usability/chi-sig-usability.pdf  
[Accessed 27 September 2023].

Slant, 2021. *Slant - What are the best dependently typed languages?.* [Online]   
Available at: https://www.slant.co/topics/10466/~dependently-typed-languages  
[Accessed 27 September 2023].

Taylor, K., n.d. *HiTechNectar.* [Online]   
Available at: https://www.hitechnectar.com/blogs/cross-platform-app-development-language/  
[Accessed 27 September 2023].

Tourville, G., n.d. *Duck Programming Language Homepage.* [Online]   
Available at: http://ducklang.org/designing-a-programming-language-i  
[Accessed 27 September 2023].

1. Commenting a piece of code, is placing human readable descriptions inside of the program, detailing what the code is doing and why. [↑](#footnote-ref-1)
2. Refinement Types are specialized data types, which are able to enforce additional constraints and rules on the values it can hold, ensuring the content adheres to specific conditions and/or criterion. [↑](#footnote-ref-2)
3. An SMT Solver is an algorithmic approach of determining whether a mathematical formula is satisfiable, in the scope of MPIR it can be used to statically determine whether a branch control will be called at compile-time. [↑](#footnote-ref-3)
4. Example Declaration of a Refinement Type as a formal declaration of a password specification. The value must be of length greater than 8 and contain at least 1 symbol. Uses the L(s) and S(s) functions declared below it. [↑](#footnote-ref-4)
5. LL(1) means that a CFG can be parsed using a top-down technique with 1 symbol lookahead. LL stands for Left-to-right, Leftmost derivation. [↑](#footnote-ref-5)