# The Typma Programming Language Senior COMPS Project

Sacha Peterson
speterson@oxy.com
Occidental College

## 1 Introduction

An import feature of modern programming languages is a type system. The purpose of a type system as described in Pierce (2002) [8] is to help the programmer write a program and find errors before the code is used. However, many programming languages fall short of an ideal type system, allowing for a variety of situations that are not intended by the programmer, causing bugs and breaking code. To expose the faults of accepted programming languages, the Typma programming language project sets out to highlight those issues by designing and writing a formally verified interpreter for a new language that takes type system problems to their logical extreme as a parody.

The specific problems of type systems that the Typma programming language project is focusing on are issues involving a dynamic type system. A dynamic type system is when the types are determined at runtime as opposed to a static type system, which determines types at compile time. Examples of popular programming languages that have a dynamic type system are Python and Javascript. Dynamic type systems allow for greater flexibility than static type systems and are becoming popular as a result as said in Pradel (2014) [10]. However, this flexibility can cause problems with programs misusing the type system for unintended results.

This flexibility can be seen in some basic Javascript expressions that are allowed by the complier, but do not make any logical sense. For example, true + true evaluates to 2 in Javascript. Despite the fact that true is a boolean and addition is an arithmetic operation, Javascript finds a way to make the expression work by changing true to the number 1 and if the boolean was false, it would be changed to 0. Another example is the expression 70 - "1" which evaluates to 69. Once again, Javascript does the impossible and converts "1" to the number 1 because the string happens to be the character. While expressions like these are funny, if a programmer accidentally performed an operation like the ones above, then no error would be present to point out the problem, which makes the bug harder to find and remove.

The main feature of the Typma programming language is to exaggerate the issues of a dynamic type system by allowing expressions to evaluate regardless of what data types are being used. In Typma it is possible to write programs with expressions like adding a boolean and a string together or performing a less than operation on a natural number and a string. These nonsense expressions in Typma can be avoided by only using the useful features and expressions but this can be easily missed as mistakes are hard to avoid and find when type errors are not present to help debug code.

## 2 Technical Background

The Typma programming language is a part of a family of languages called esoteric programming languages. The idea of an esoteric programming language is having a programming language that is not meant to be used, and either serves "to entertain, to be beautiful, or to make a point" as described in Morr (2014) [7]. In the context of the Typma programming language, Typma is meant to entertain programmers by allowing expressions that do not make sense and are hard to debug. Moreover, Typma is meant to make a point about how some of Typma's problems with its type system exist in popular languages and if these problems are not okay in Typma, then they are not okay in any programming language. Typma can be referred to as a joke programming language due to the whimsical nature of the language to further distinguish Typma and other gag languages from more serious esoteric programming languages.

Features of functional programming that are described in Clarkson (2021) [3] are important aspects of the Typma project. In functional programming, algebraic data types are type variants that can contain both sum types and product types. A sum type is "a value of a variant [that] is formed by one of the constructors" and product types are "tuples or records, whose values have a sub-value from each of their component types" as written in Clarkson (2021) [3]. To do case analysis on different variants of an algebraic data type, pattern matching is a way to see what variant of a type an input is, and what to do with it when the right pattern is found. This code structure prevents missing a variant, because the type checker will throw an error at compile time if any variants are not accounted for.

The Typma programming language is an extension of the IMP language, an imperative language with basic functionality as seen in *The IMP Language* [11]. IMP features sup-

port for arithmetic expressions, boolean expressions, variable assignment, if statements, and while statements. IMP does not have a type system because booleans and boolean expressions are limited to conditional expressions in if statements and while loops. The purpose of IMP is to teach semantics of programming languages and is not meant to be used for actual programming.

The Typma codebase makes use of a lexer, parser, and interpreter. A lexer is a program that takes in a text and outputs the text represented by tokens depending on what the text is. A parser is a program that takes in the tokens from the lexer and outputs an abstract syntax tree (AST). An abstract syntax tree is a data structure that represents a program, where each node represents a part of the code. The interpreter will take in the tree and evaluate the code to the specifications of the Typma language, outputting any print statements from the code.

The interpreter is written in a tool called a proof assistant, a special kind of programming language that supports proofs along side normal code. Pierce (2021) [9] describes proof assistants as "hybrid tools that automate the more routine aspects of building proofs while depending on human guidance for more difficult aspects. Widely used proof assistants include Isabelle, Agda, Twelf, ACL2, PVS, and Coq, among many others". This project uses the Coq proof assistant, while any of the other proof assistants could have been used for this project. Coq was chosen because of the excellent guide for Coq in Pierce (2021) [9], which contains exercises on IMP that make for a solid starting point for Typma. Coq, according to Pierce (2021) [9], has been used, "as a platform for modeling programming languages, it has become a standard tool for researchers who need to describe and reason about complex language definitions. It has been used, for example, to check the security of the JavaCard platform, obtaining the highest level of common criteria certification, and for formal specifications of the x86 and LLVM instruction sets and programming languages such as C". Pierce (2021) [9] also writes that Coq has been used "As an environment for developing formally certified software and hardware, Coq has been used, for example, to build CompCert, a fully-verified optimizing compiler for C".

The Typma programming language is written in big step semantics. According to *The IMP Language* [11], big step semantics are when evaluation steps are combined into a larger step for evaluation. Another evaluation method is small step, when each evaluation step is evaluated one at a time. The difference between big and small step semantics can be seen with an addition expression $1 + 2 + 4$. In big step, this would be one compound step, with a sub step evaluating $1 + 2$, then adding that to $4$. In small step, $1 + 2$ would be evaluated then the result $3$ would be added to $4$, $3 + 4$. An implementation of big or small step is trivial and Typma would function the same regardless of whether big or small step was used for evaluation.

## 3 Prior Work

There are a great number of esoteric languages that have been created with some similarities and differences to the Typma programming language in both function and purpose. Brainfuck is a well-known esoteric programming language that was designed to have the smallest amount of syntax symbols and the smallest size of compiler of any programming language as described in Morr (2014) [7]. Another famous esoteric programming language is Malbolge, a language that was designed to be near-impossible to both understand and use also seen in Morr (2014) [7]. These are famous esoteric programming languages but they are very different from the Typma programming language project. Currently, there are no esoteric programming languages that have the same or similar concept to Typma but the idea of taking a feature to its logical extreme can be found in an esoteric language called J.A.V.A.

J.A.V.A., short for Just Another Verbose Annoyance, is a language that can be described as an "object disoriented language" found on *J.A.V.A.* [6]. The point of this programming language is to exaggerate the problems of object-oriented programming and more specifically, the issues with Java's class system. J.A.V.A. adds several extra steps to creating a class compared to Java. The first step is creating a class, but that requires a class factory. Creating a class factory requires a class factory provider. The only implementation of a class factory provider is a ClassFactoryProviderSingleton, which contains a static instance of a singleton. By adding all of these steps to making a class, J.A.V.A.'s type system becomes very complicated as a result. If a programmer wanted to print something in J.A.V.A., they would have to call the StatementBuilder object just to make a Statement, then an ExpressionBuilder object to make an Expression all just to get a string into a print statement. Although the approach is different, J.A.V.A. and Typma both make fun of awful type systems, with J.A.V.A. forcing the use of a large number of objects to achieve a simple task. The main feature of J.A.V.A. is focused on taking Java's implementation of object oriented programming to its logical extreme similar to how Typma takes a dynamic type system issues to their logical extreme.

The Typma programming language project takes an indirect approach to expose the problems of a bad type system. On the other hand, TypeDevil from Pradel (2014) [10], takes a direct approach to show the problems of Javascript's dynamic type system. TypeDevil is a benchmark that generates warnings if inconsistent types are found. TypeDevil is tool that is meant to be used to help real programming projects, whereas Typma is not meant to be used for any se-

rious programming project of any kind. Also, TypeDevil is focused on Javascript but Typma is related to any language with a faulty dynamic type system.

## 4 Ethical Considerations

Computer science as a field has a massive bias towards the English language with the use of English for keywords in programming languages. Python, C, C++, Java, Javascript, Haskell, OCaml, and Rust all have English as the language of the language. Typma will be no different from these mainstream languages by also using English keywords. While this poses no issues for native English speakers, people for whom English is a second language or people who do not know know any English are at a major disadvantage when it comes to becoming an experienced computer scientist.

In Veerasamy (2014) [12], researchers decided find if computer science higher education "students' English knowledge of programming keywords and their programming abilities are related with each other." This paper outlined a methodology to test this by giving students an English language test and a computer science test to see the performance of the students in each fields. The students would not be native English speakers and some students would take the computer science course in English and the others would take the course in their native language. This experiment was not tested, the paper set out to outline a study not perform it so there are no results to prove the claim. The framework could be used in an experiment in the future to prove the claim of non-English speakers having more success learning computer science in their own language. For this paper, the discussion does not extend to changing the keywords, just focusing on the language of the education.

For Idris (2018) [5] a test similar to the one prosed in Veerasamy (2014) [12] was conducted. A study was done at an Arabic-speaking university to see if English language ability was correlated to higher performance in computer science courses. The study also looked at whether lecturers, labs, mathematics background and logical thinking affect programming skills to see if those factors are more impactful than language. The study found that there is a moderate correlation between programming ability and English comprehension, showing a need for computer science courses in other languages. The study concluded that a language should be created that is written in Arabic and is designed to teach programming. This could be used for classes taught in Arabic to help students who are unfamiliar with English learn computer science.

In addition, Guo (2018) [4] performed a survey to see how the English barrier affected learning computer science in the context of online learning as opposed to the university setting of Idris (2018) [5]. Moreover, this survey received 840 responses from 86 countries and 74 native languages, different from just studying one language. The survey consisted of eight questions, such as asking the participant what their native language was, level of English fluency to describe the participant's background. There were three questions directed to non-native English speakers about difficulties with in programming because of the language barrier, improvements to instructional materials, and how programming has affected English proficiency. One question was directed to native English speakers about observations they have on non-native English speaker's difficulties with programming. The study found responses discussing problems like non-native English speakers writing unintelligible code, documentation and tutorials containing cultural references and slang difficult for non-native speakers, and learning computer science and English at the same time being difficult.

The issue of the English barrier to the Typma programming language project and computer science in general is a real problem that gives native and fluent English speakers an advantage in computer science. The Typma project is unable to address this issue because my only fluent language is English. Typma could have alternate parsers for other languages, but that does not change that the concepts would still be hard to convey because English is at the center of programming language design. The only thing I could do is write a version of Typma in Japanese, as I have six years of Japanese language education as well as a minor in Japanese language studies. My basic understand of Japanese language could allow me to write a version of Typma that is understood by a Japanese native speaker, however I am not sure how to evaluate how well this version would make Typma more accessible as I do not know anyone who is both learning computer science, has Japanese as their native language, and either does not know or has only a limited understanding of the English language. Moreover, this could make the project out of scope because the focus is on the semantics, not the lexing side of programming languages. While the problem of the English barrier is an interesting and complex problem that could exclude any non-English speaker from getting involved with the field of computer science, the scope of the Typma programming language project is far removed from practical usage and as a result has no connection to any ethical concerns.

## 5 Methods

The first step of the Typma programming language project is writing the formal semantics of the language. These semantics detail every situation that can happen in Typma and what the result is for that situation. Typma is an extension of IMP that adds nonsense expressions to

the specifications of the brand new system of math Typma calculus. The supported statements of Typma are skip, variable assignment, print, if, and while statements. Typma supports arithmetic expressions, boolean expressions, and comparison expressions. Typma supports three basic literal data types: natural numbers, booleans, and strings with every expression supporting every data type.

Metavariable for Typma values: $t \in$ typma, $t$ ::=

- $n \in \mathbb{N}$
- $b \in \mathbb{B}$
- $s \in$ string

Natural numbers were chosen over integers because natural numbers are easier to work with in the proof assistant and the language would be fundamentally the same with integers. Ironically, Typma does not have a type system because natural numbers, booleans, and strings are different cases of a single main type, the Typma data type. This means that although Typma is a parody of a dynamic type system, Typma is not an implementation but an approximation. Here is the syntax of Typma written in type theory notation:

Expressions: $e \in$ exp, $e$ ::=

- $n$
- $b$
- $s$
- $x \in$ variable name
- $e_1 + e_2$
- $e_1 - e_2$
- $e_1 \times e_2$
- $e_1 \div e_2$
- $e_1 == e_2$
- $e_1 < e_2$
- $e_1$ && $e_2$
- $e_1 \mid\mid e_2$
- $!e$

Commands: $c \in$ com, $c$ ::=

- SKIP
- $x := e$
- $c_1; c_2$
- IF $e$ THEN $c_1$ ELSE $c_2$
- WHILE $e$ DO $c$
- PRINT $e$

Here are the semantics of Typma:
$\Downarrow$ exp $\subseteq$ (exp $\times$ Store) $\times$ $t$
$\Downarrow$ com $\subseteq$ (com $\times$ Store) $\times$ Store

Expressions:

$$\frac{}{\langle \sigma, t \rangle \Downarrow t}$$

$$\frac{\sigma(x) = t}{\langle \sigma, x \rangle \Downarrow t}$$

$$\frac{\langle \sigma, e_1 \rangle \Downarrow t_1 \qquad \langle \sigma, e_2 \rangle \Downarrow t_2 \qquad t = t_1 + t_2}{\langle \sigma, e_1 + e_2 \rangle \Downarrow t}$$

$$\frac{\langle \sigma, e_1 \rangle \Downarrow t_1 \qquad \langle \sigma, e_2 \rangle \Downarrow t_2 \qquad t = t_1 - t_2}{\langle \sigma, e_1 - e_2 \rangle \Downarrow t}$$

$$\frac{\langle \sigma, e_1 \rangle \Downarrow t_1 \qquad \langle \sigma, e_2 \rangle \Downarrow t_2 \qquad t = t_1 \times t_2}{\langle \sigma, e_1 \times e_2 \rangle \Downarrow t}$$

$$\frac{\langle \sigma, e_1 \rangle \Downarrow t_1 \qquad \langle \sigma, e_2 \rangle \Downarrow t_2 \qquad t = t_1 \div t_2}{\langle \sigma, e_1 \div e_2 \rangle \Downarrow t}$$

$$\frac{\langle \sigma, e_1 \rangle \Downarrow t_1 \qquad \langle \sigma, e_2 \rangle \Downarrow t_2 \qquad t = t_1 == t_2}{\langle \sigma, e_1 == e_2 \rangle \Downarrow t}$$

$$\frac{\langle \sigma, e_1 \rangle \Downarrow t_1 \qquad \langle \sigma, e_2 \rangle \Downarrow t_2 \qquad t = t_1 < t_2}{\langle \sigma, e_1 < e_2 \rangle \Downarrow t}$$

$$\frac{\langle \sigma, e_1 \rangle \Downarrow t_1 \qquad \langle \sigma, e_2 \rangle \Downarrow t_2 \qquad t = t_1 \&\& t_2}{\langle \sigma, e_1 \&\& e_2 \rangle \Downarrow t}$$

$$\frac{\langle \sigma, e_1 \rangle \Downarrow t_1 \qquad \langle \sigma, e_2 \rangle \Downarrow t_2 \qquad t = t_1 || t_2}{\langle \sigma, e_1 || e_2 \rangle \Downarrow t}$$

Commands

$$\text{SKIP}\frac{}{\langle \sigma, \text{SKIP} \rangle \Downarrow \sigma}$$

$$\text{ASSIGN}\frac{\langle \sigma, e \rangle \Downarrow t}{\langle \sigma, x := e \rangle \Downarrow \sigma[x \mapsto t]}$$

$$\text{SEQ}\frac{\langle \sigma, c_1 \rangle \Downarrow \sigma' \qquad \langle \sigma', c_2 \rangle \Downarrow \sigma''}{\langle \sigma, c_1; c_2 \rangle \Downarrow \sigma''}$$

$$\text{IF-T}\frac{\langle \sigma, e \rangle \Downarrow \text{TRUE} \qquad \langle \sigma, c_1 \rangle \Downarrow \sigma'}{\langle \sigma, \text{IF } e \text{ THEN } c_1 \text{ ELSE } c_2 \rangle \Downarrow \sigma'}$$

$$\text{IF-F}\frac{\langle \sigma, e \rangle \Downarrow \text{FALSE} \qquad \langle \sigma, c_2 \rangle \Downarrow \sigma'}{\langle \sigma, \text{IF } e \text{ THEN } c_1 \text{ ELSE } c_2 \rangle \Downarrow \sigma'}$$

$$\text{WHILE-F}\frac{\langle \sigma, e \rangle \Downarrow \text{FALSE}}{\langle \sigma, \text{WHILE } e \text{ DO } c \rangle \Downarrow \sigma'}$$

$$\text{WHILE-T}\frac{\langle \sigma, e \rangle \Downarrow \text{TRUE} \qquad \langle \sigma, c \rangle \Downarrow \sigma' \qquad \langle \sigma', \text{WHILE } e \text{ DO } c \rangle \Downarrow \sigma''}{\langle \sigma, \text{WHILE } e \text{ DO } c \rangle \Downarrow \sigma''}$$

The general rules for all possible combinations of data types in an operation is laid out a new system of math created just for Typma called the Typma calculus, which supports the operations addition, subtraction, multiplication, division, equality, less than, and, or, and not. Every combination of natural number, boolean, and string has been accounted for with an intended result for each operation. The main concept for the design of the Typma calculus was that each data type has a converted form when it needs to be treated as a different data type. When a natural number is being put in an && operation with a boolean, like 3

&& true, the 3 is converted into a form that makes more sense for an && operation. The 3 is turned into the boolean true because all numbers greater than 0 are true in the conversion. This is based off of C booleans being represented with 0 as false and anything greater than 0 as true. The example expression would evaluate to true because the evaluation turns 3 && true into true && true.

There is a conversion for each type into the other for situations like this. If a natural number needs to be a string, 0 would become "no succ" and anything greater than 0 would become "succ." This is a joke on the church numeral representation of natural numbers which is discussed in Pierce (2021) [9], as 0 is the base and everything else is just a successor of 0. When a boolean needs to be changed to a natural number, false becomes 0 and true becomes 1 similar to the opposite conversion based on C. Converting boolean to string is just the word as a string, like false becoming "false" and true becoming "true." The conversion from string to natural number is just the length of the string and string to boolean is just the empty string becoming false and anything else becoming true. These conversions are meant to be arbitrary and confusing if a programmer accidentally writes a nonsense expression, to make debugging as annoying as possible. What makes Typma even more arbitrary is that what type gets converted in an operation has specific to each operation. For example, many of the boolean expressions convert natural numbers into booleans, but only the ! operator keeps the natural number as a number and performs a factorial operation on the number. Data type conversions also take place when combined with certain command statements. All print statements in Typma only output strings, so printing a number will only print out "no succ" or "succ," making print debugging more frustrating. Moreover, conditionals in if statements and while loops will convert an expression into a boolean because booleans are necessary for a conditional to function. Basically, the outcome of each combination was decided individually on a whim based on inconsistent rules to make the system as arbitrary and confusing as possible.

A special case of the Typma calculus is division because normal math does not allow for division by zero. The lack of a result for these operations allows the Typma calculus to fill in the gap with an evaluation that makes programming more annoying. Every operation that can evaluate to a division by zero, including division by false or an empty string, will evaluate to the copy pasta from *[Copypasta] FitnessGram Pacer Test* [1]. In addition to not making any logical sense, a correct evaluation for division by zero can be cause for some very buggy code if a programmer ever accidentally did it. Other languages may throw an error but no operation is out of the reach of Typma's ability to evaluate.

The next step of the project is to create an interpreter program that runs Typma code. The implementation of the lexer, parser, and command line arguments are written in OCaml, but the evaluation functions are written in Coq, a proof assistant as seen in Pierce (2021) [9]. The purpose of the proof assistant is to provide proofs for the code to show that they are correct, or do not have any unintended outcomes that cause bugs in the evaluation. In the proof assistant, the evaluation functions are defined along with an inductive proposition for each function. An inductive proposition is a data type that contains each possible case that comes as a result of running the respective function. To show the functions are correct, a proof for each function is completed that shows that the functions and the inductive propositions are bijectively reflexive. This means that if the function is correct, that implies the inductive proposition is correct and vise versa. Because of these definitions and proofs, this will allow for a greater degree of representing the semantics of Typma correctly as opposed to directly writing the evaluation functions in OCaml. Just relying on test cases to show the correctness of the evaluation is not sufficient if I am not able to find every bug in the code. Instead of using Coq, I could use a program like PLT Redex, to implement Typma, but Coq is simple to integrate with an OCaml project and achieves a better result with less steps. PLT Redex does not prove that the interpreter follows the formal semantics whereas proofs in Coq do.

The Coq file, containing the data type for syntax and evaluation functions is extracted into an OCaml file to be ran with the rest of the codebase. The extraction process occurs when the codebase is complied or can be complied separately from the rest of the codebase. The proofs and inductive propositions stay in the proof assistant because they are not needed to run a Typma program, but they are checked to be correct when the codebase is complied. The extraction process is a general tool I used that is built into Coq as a library and can be used for extraction to OCaml, Haskell, and Scheme (cite the website). I could have used any of those languages for my codebase, but I chose OCaml because I am familiar with the parsing tools for OCaml and I have never used either Haskell or Scheme.

While the extraction process is automated, sometimes the extraction generates code that does not compile in OCaml. For example, because strings were used in the Coq file, the generated OCaml file had an error confusing functions from the boolean library with functions from the string library. To fix this, I created a wrapper type in the Coq file for strings called Sus and replaced every use of a string or a string library function with the Sus version. This prevented the extraction from generating the bugged code and was able to compile.

Two key features of Typma are not able to be represented in the proof assistant code. The first feature that had to be added outside of the proof assistant was printing to the console. Coq does not have any print statements so the rep-

resentation of Typma print statements in Coq is just an expression that evaluates inside of the word print. The console print is added to the evaluation function for command statements during extraction as OCaml has print statements.

The other feature that is not present in Coq is an infinite while loop. A representation of a while loop has a case when a while loop is infinite, leading to arbitrary recursive calls which Coq does not support. Pierce (2021) [9] writes that, "since Coq is not just a functional programming language but also a consistent logic, any potentially nonterminating function needs to be rejected". Arbitrary recursive calls would allow for a recursive function that proves false, "which would be a disaster for Coq's logical consistency" as seen in Pierce (2021) [9].

In order to represent loops in Coq without arbitrary recursive calls, the command statement evaluation function has an extra number as a parameter called a gas value. When the command statement function is used, a large number is put in gas parameter. Every time the function performs a recursive call, the value of the gas is decremented, and when it reaches zero, the function returns a none value, showing that the function has ran out of gas from Pierce (2021) [9]. The case analysis of the evaluation function in the main will print out the Yoshikage Kira copy pasta upon receiving the none result from running the program similar to the result of dividing by zero from *[Copypasta] My name is Yoshikage Kira* [2]. Infinite loops are not the only way to run out of gas and sequences are also a recursive call that decrements the gas value every time one is present, so if there are enough sequences, the gas will run out.

This workaround for infinite loops using gas values is not the only method. Another way to represent loops would be the use of small step semantics as opposed to big step. This would involve a limit on the number of steps a program could perform in the Coq file, which is similar to the gas value, and this could be reached by writing a program that does enough recursive calls to run out of gas. Either of the methods ends up being the same and big step was the method chosen for this project.

A way that gas values can be avoided is using coinductive data structures as seen in Xia et al. (2020) [13]. Coinduction was used to write an interpreter for IMP in big step without a gas value for while loops and since Typma is an IMP extension, coinduction could be also used for Typma. However, coinduction is complicated and the interpreter would evaluate the programs to the same results either way so coinduction was out the scope of this project.

With the finished extracted syntax and evaluation, the next step is to write a parser takes in a raw text file and converts it into an AST for the evaluation functions. I used the libraries OCamllex and Menhir which provided a straightforward tool to set up a parser in OCaml. The parser functions the same as a parser from a normal programming language, the intentionally bad and confusing aspects of Typma as a programming language are limited to the evaluation step so the parser is just a normal parser. The only notable aspect of the parser is that the algorithm in the parser had to account for any data type in any place. Basic paring algorithms may only have numbers accounted for in an arithmetic expression and booleans in boolean expressions, but because Typma allows any type in any expression, the parsing rules for all types are connected to each other so the parser is able to detect numbers in a boolean expression or strings in an arithmetic expression.

It is possible to write a parser that does not use a lexer, but manually parsing code is more difficult and time consuming. Moreover, the libraries OCamllex and Menhir are set up for a parser and lexer, so committing to writing a manual parser would require spending time to write code that is already written by those libraries. The focus of the Typma programming language project is on the evaluation step of the codebase, so there is no need to re-invent a parser that will function the same as any other parser.

Many of the representations of the Typma programming language could be represented with objects as opposed to algebraic data types. For example, the contents of the syntax are all declared as types with different variants for each kind of syntax. Objects are another way to represent the syntax of the Typma programming language. The math expression type could instead be a math expression class, with the variants integer, variable, and math expression instead being subclasses. While objects can be used to represent the syntax, this is not optimal when compared to an algebraic data type. The code for an algebraic data type is a fraction of the code that would be needed to make classes and subclasses. In addition to having significantly more code to accomplish the same task, the code for evaluation would be unable to take advantage of pattern matching on the different patterns for commands and expressions in Typma. Using algebraic data types is less work and less code than objects for this project.

The final step of the project is coding the command line interface to run a Typma program from a file. To run Typma programs, a basic command takes the name of a file, finds the file, and runs the code through the interpreter. This is the part of the codebase that connects the parser to the evaluation functions, serving as the main file of the project. When a program is ran, the command line displays what the program got parsed as, and if the program has any print statements, their output is displayed below.

## 6 Evaluation Metrics

The metrics of evaluation for the Typma interpreter include some basic test cases. This small set of test cases show basic examples of every command statement being

used correctly along with various expressions that show if the Typma calculus is implemented correctly. If the test cases are ran and they do not cause any runtime errors in the codebase and give the correct result, then the interpreter works for those cases. For example, the test6.typ file, contains a basic while loop printing an incremented number five times, is run with the command "dune exec ./bin/main.exe –-typma tests/test6.typ." When the command is ran, the correct output would display what the parser parsed the code as. A toString version of the AST that should be equal to what was written in the file. Then the print statements in the file would print out the numbers. If the output is what is expected, then the test case passed. These test cases are still not enough to show that the interpreter is working in every case so additional evaluation is needed.

The evaluation of Typma's evaluation functions is built into the design of being written in the proof assistant. A proof that the evaluation function is equivalent to the inductive proposition representing the formal semantics is evidence that the codebase's evaluation functions are accurate to the formal semantics. Just relying on test cases is not rigorous enough to show that the Typma codebase succeeded in representing the semantics of Typma. If the proofs are cleared by the Coq complier, that is enough evidence to show that the task has been performed correctly. This assumes that the Coq compiler is correct, but that is a safe assumption because Coq "has been under development since 1983 and that in recent years has attracted a large community of users in both research and industry" as seen in Pierce (2021) [9]. One of the proofs is separate from showing the correctness of the evaluation function but instead shows if Typma is deterministic. A programming language is deterministic if the same inputs will always yield the same results.

The evaluation functions for expressions and commands have proofs but all of the Typma calculus functions are not included in the proof checking. If proofs were done for those functions, they would be completed in one line with the reflexivity tactic. This means that the proof is trivial and the proof would be superfluous because the correctness of the function is given. Typma calculus functions are a direct translation of the system of math and therefore are inherently correct.

The evaluation that the parser has no bugs is shown with a fuzzing program. The fuzzer generates a number of random syntactically valid Typma programs and if all of them are ran through the parser without any errors that means the parser is correct. These programs do not need to be evaluated because the evaluation is already proven to be correct. The fuzzer can be ran with a command in the command line interface, with an option for how many random programs to generate. It might be possible to generate random programs that evaluate correctly according to a given specification but that would become program synthesis which is out of scope for this project. If all of the above conditions are met, then the project has succeeded in implementing Typma in a formally verified interpreter.

## 7 Results and Discussion

The Typma programming language project was able to demonstrate the above metrics for evaluation. The basic test cases give the desired result, the proofs were proven to be correct in Coq, and the fuzzer correctly parsed up to 10,000 iterations at a single time. With this evidence, the interpreter has succeeded and is a correct implementation of the Typma programming language. Because the determinism proof has correct, Typma is a deterministic programming language.

The main purpose of Typma programming language project is to parody dynamic type systems that allow for nonsense expressions. Typma is not meant to be used as an actual language, but sets out to bring attention to flaws present in regular dynamically typed languages. Although normal programs can be written in Typma, it is more frustrating to write useful programs because a programmer would need to be hyper aware of not making mistakes where a good programming language would throw errors to stop silly mistakes. While Typma's intentionally bad type system is ridiculous, many accepted programming languages are not as sound as one would think.

## References


[1] *[Copypasta] FitnessGram Pacer Test*. Sept. 2016. URL: https://www.twitchquotes.com/copypastas/2029.

[2] *[Copypasta] My name is Yoshikage Kira*. Sept. 2019. URL: https://www.twitchquotes.com/copypastas/3451.

[3] Clarkson, Michael R. *OCaml Programming: Correct + Efficient + Beautiful*. https://cs3110.github.io/textbook/cover.html. 2021.

[4] Guo, Philip J. "Non-Native English Speakers Learning Computer Programming: Barriers, Desires, and Design Opportunities". In: *ACM* (Apr. 2018).

[5] Idris, Mrwan Ben. "The Correlation between Arabic Student's English Proficiency and Their Computer Programming Ability at the University Level". In: *West Virginia University* (Mar. 2018).

[6] *J.A.V.A.* https://esolangs.org/wiki/J.A.V.A. June 2020.

[7] Morr, Sebastian. “Esoteric Programming Languages: An introduction to Brainfuck, INTERCAL, Befunge, Malbolge, and Shakespeare”. In: *Braunschweig University of Technology* (Dec. 2014).

[8] Pierce, Benjamin C. *Types and Programming Languages*. The MIT Press, 2002.

[9] Pierce, Benjamin C. *Software Foundations Volume 1: Logical Foundations*. https://softwarefoundations.cis.upenn.edu/lf-current/index.html. Aug. 2021.

[10] Pradel, Michael. “TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript”. In: *Electrical Engineering and Computer Sciences University of California at Berkeley* (2014).

[11] *The IMP Language*. 2021.

[12] Veerasamy, Ashok Kumar. “Teaching English Based Programming Courses to English Language Learners/Non-Native Speakers of English”. In: *RMIT University* (2014).

[13] Xia, Li-yao et al. “Interaction trees: representing recursive and impure programs in Coq”. In: *Proceedings of the ACM on Programming Languages* 4.POPL (Jan. 2020), pp. 1–32. DOI: 10.1145/3371119. URL: https://doi.org/10.1145%2F3371119.