The Typma Programming Language Project COMPS Proposal

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) [7] is is 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 creating a language that has a bad type system on purpose.

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) [9]. However, this flexibility can cause problems with programs misusing the type system for unintended results.

Javascript is a great example of how a dynamic type system can cause problems for the programmer. Pradel (2014) [9] shows examples of code that leads to inconsistent types. A simple string concatenation program has the result start with “undefined” because the initial variable was not given a type.

```
var str;
for (i in seqs) {
  str += seqs[i].source;
}
```

A small issue like this can have a big impact on a program and could hurt the functionality. If an undefined variable was in a bank system, then someone’s name could be “undefinedJoseph Joestar,” making it hard for someone to access their money. A problem like this would be exaggerated in Typma. If this code was in a Typma program, at run-time the interpreter would randomly decide if the data will be represented as a string, integer, or boolean. This outcome could change every time the code is ran depending on what random number generation decides.

The purpose of the Typma programming language is to exaggerate the issues of a dynamic type system by making the types so dynamic to the point that the compiler will make any data type work in any situation by using random number generation to resolve the ambiguous situation. The random typing can be avoided by specifying every instance of typed data but this is can be easily missed as type errors do not exist in Typma.

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) [5]. In the context of the Typma programming language, Typma is meant to entertain programmers by using an over-designed method of type specification with comedy and pain being achieved when the programmer forgets to specify a data type leading to random results from the Typma program. 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.

A component of the Typma programming language project is type theory. As described in nLabauthors (2022) [6], ”type theory is a branch of mathematical symbolic logic, which derives its name from the fact that it formalizes not only mathematical terms – such as a variable x, or a function f – and operations on them, but also formalizes the idea that each such term is of some definite type.” For example, x can have the type $\mathbb{N}$ for natural number which is different from the type of the function f $\mathbb{N} \to \mathbb{N}$, this represents a function that takes in a $\mathbb{N}$ and returns a $\mathbb{N}$. Type theory is the basis for any programming language's type system, and Typma is no exception. The notation and definitions of type theory will be used to formalize the semantics of the Typma programming language. A proof assistant will be used which as described in Pierce (2021) [8] is a "hybrid [tool] that [automates] the more routine aspects of building proofs while depending on human guidance for more difficult aspects."

An important aspect of the Typma project are features of functional programming that are described in Clarkson (2021) [1]. 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) [1]. 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 cornel notes *The IMP Language* [11]. IMP features support for arithmetic expressions, boolean expressions, variable assignment, if statements, and while statements. 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 Typma programming language can be evaluated either in big step or small step semantics. According to *The IMP Language* [11], big step semantics are when evaluation steps are combined into a larger step for evaluation and small step semantics are 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 $1 + 2 + 3$. In big step, this would be one compound step, with a sub step evaluating $1 + 2$, then adding that to 3. In small step, $1 + 2$ would be evaluated then the result of 3 would be added to 3, $3 + 3$.

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) [5]. 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) [5]. These are famous esoteric programming languages but they are very different from the Typma programming language project. There are two esoteric programming languages that are relevant to the Typma programming language project. J.A.V.A. is focused on an obtuse object system which also leads to an obtuse type system and SimPPLe which is focused on random number generation messing with variables.

J.A.V.A., short for Just Another Verbose Annoyance, a language that can be described as an object disoriented language found on *J.A.V.A.* [4]. The point of this programming language is to exaggerate the problems of object-oriented programming and more specially, the issues with Java's class system. J.A.V.A. adds several extra steps to creating a class compared to Java. These steps are creating a class requires a class factory, which requires a class factory provider, which 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 have awful type systems, with J.A.V.A. forcing the use of a large number of objects to achieve a simple task. Typma does not have objects at all but requires all instances of a value with type be specified or the interpreter with randomly decide what the type is at runtime. As a result, correct Typma is verbose with the type specifiers but not as verbose as a cascade of object instantiations.

SimPPLe, short for Simple Probabilistic Programming Language, is a probabilistic programming language that can be found on *SimPPLe* [10]. SimPPLe programs will produce different results every time they are ran because the language is based on random number generation. Any variable that is declared will be assigned a random number, making programs useless as there is no control of what values variables have in SimPPLe. As the name suggests, this language is simple as there is only support for basic math operations. Typma code that is written without type specifications is similar in function to SImPPLe, the difference is that Typma randomizes variables based on what type they could be, meaning there are only three ways each variable can be interpreted: as, integer, string, and boolean, whereas SimPPLe variables can be any integer, but no other data type. Typma also expands this random nature to a language with high level statements like an if statement or a while loop for more complicated outcomes.

The Typma programming language project takes indirect approach to expose the problems of a bad type system. On the other hand, TypeDevil from Pradel (2014) [9], 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 serious 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) [3] 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 if 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) [2] 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) [3]. 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 and 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 this as I do not know anyone who is learning computer science with Japanese as their native language. Moreover, this could make the project out of scope because the focus is on the semantics, not the parsing side of programming languages. While the problem of the English barrier is not able to be solved for the Typma project, if times allows and the project is finished early, I may implement a Japanese version of Typma.

5 Methods

The first step of the Typma programming language project is writing the formal semantics of the language. Not every situation regarding Typma's type system has been decided at this time, and the creation of Typma's formal semantics would specify all details of the language. At a base level, Typma is an extension of IMP, adding functions, making small changes to syntax, and adding a type system with Typma's key features. The supported statements of Typma are variable assignment, print statements, if statements, while statements, and function statements. Typma supports arithmetic expressions, boolean expressions, and comparison expressions. In Typma, integers, strings, and booleans can be specified with Typma's specification operators. These specification operators were designed to look as annoying as possible.

```
INT[10];

STR[Hello];

BOOL[true];
```

Type errors do not exist in Typma because the interperter will resolve any lexical ambiguity or deliberate type errors as a result of improper usage of specification operators with random interpretations. For example:

```
print(10);

print(INT[10]);
```

Technically, print statements only accept strings for data, and if there was no evaluation to convert an integer to a string, this statement would throw a type error because there is an integer in a print statement. However, Typma's implementation of dynamic typing with random number generation, will decide what the integer 10 could be as a string amongst some possibilities. Is the integer an ASCII value? Then the interpreter will print out the character 'A' with the ASCII value of 10. Is the integer just a string "10"? Then the interpreter will print out the string "10". This system is how ambiguity is solved when either no type is specified or the wrong type is specified.

To run Typma programs, a basic command line interface will take in a file to be ran through the Typma codebase. If the program has any print statements, the output will be displayed on the command line. There will be a different command for evaluation in big step or small step semantics. The codebase of Typma will contain a lexer, parser, and interpreter that will take in Typma code as in input and output the results of the Typma program as per the specifications of Typma semantics.

The implementation of the lexer, parser, and command line arguments will be written in OCaml, but the interpreter will be written in Coq, a proof assistant as seen in Pierce (2021) [8]. In the proof assistant, an inductive data-type will be defined for syntax, as well as both a function that type-checks the syntax and an inductive proposition that specifies the typing judgement for the syntax. A proof will be written that shows that the inductive proposition and the function are equivalent. The evaluation rules will be defined as both a recursive function and an inductive proposition. Another proof will to show that the evaluation function and the inductive proposition are doing the same thing for each rule, providing evidence that the codebase is a sound representation of Typma's formal rules. The data-type for syntax, type-checking function, and evaluation function will extracted into an OCaml file to be ran with the rest of the codebase. The proofs and inductive propositions stay in the proof assistant and are not ran when a Typma program is being evaluated. Because of these definitions and proofs, this will allow for a greater degree of representing the formal semantics of Typma correctly as opposed to directly writing the evaluation functions in OCaml. Instead of using Coq, I could use a program like PLT Redex, to prove the correctness of my Typma implementation, but Coq is simple to integrate with an OCaml project and achieves the same result with less steps.

It is possible to write a parser programming that does not use a lexer, but manually parsing code is more difficult and time consuming because reasons. 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.

The interpreter will support an option to evaluate the code based on big step or small step semantics. Choosing big or small step does not change the outcome of the program because they are equivalent, the option is there for added detail. If time allows, concurrency maybe added to the small step version for add chaos with respect to the order that operations with unspecified types are evaluated in.

Many of the representations of the Typma programming language could be represented will 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 the different patterns for commands and expressions in Typma. Using abstract data types is less work and less code than objects for this project.

6 Evaluation

The first part of evaluating the function of the Typma codebase is if the OCaml complier type-checks the code at compile time. If the Typma codebase type-checks in OCaml, it is not a guarantee that the semantics of Typma have accurately re-created, but it does show that the types of the functions return the type that the are supposed to and that all patterns have been accounted for in any pattern matching statements.

In addition to the OCaml compiler type-checking, to check if the Type codebase is an accurate representation of the Typma programming language, there will be test cases of Typma code that will give the intended result as per the specifications of the Typma semantics. These test cases will provide evidence that the Typma codebase is a proper implementation of the Typma programming language as defined in the formal semantics of Typma. Additionally, small test cases will be present in the proof file to test evaluation of functions.

Here is an example of a test case with Typma code:

```
print(STR[10]);
```

If the Typma code base prints a 10, then the function of the code base is shown to be correct for this feature. All test cases that do not involve random outcomes will be ran with a program that checks to see if the correct result if given.

Many test cases will have a random number of outcomes, here is an example:

```
print(10);
```

This test case could print out "10" as a string, but the random typing could type it as a boolean, true like a c boolean and print the string "true", or 10 could be an ASCII value for a symbol. These random test cases would have to be run multiple times to show that the Typma code base is functioning correctly. The results of the random tests will have to be checked to see if every possibility happened in the test and if the distribution is equal, as the random nature of Typma favors no outcome and all possibilities have an equal chance of happening.

The final evaluation of the Typma code base would be the correct implementation of Typma's evaluation functions in a 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 represent the semantics of Typma. If the proofs of the type-checker and the evaluations 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) [8].

7 Proposed Timeline

Assuming no work in done directly on the project, here is the proposed bi-weekly timeline:

- Sep 1: Write out the formal semantics of Typma
- Sep 15: Complete parser/lexer of base language, the base language is an implementation of IMP with functions and uses Typma syntax
- Sep 29: Complete and prove the interpreter of base language
- Oct 13: Formally write out semantics for Typma
- Oct 27: Check with test cases, start poster
- Nov 3: Add Typma types to parser, work on poster
- Nov 15: Poster due
- Nov 17: Add Typma types to interpreter and prove them
- Dec 1: Check with test cases
- Dec 15: Project due

8 Conclusion

The Typma programming language project is a parody of dynamic type systems. Although normal programs can be written in Typma, it is more frustrating with one mistake of missing a type specifier operator causing the program to produce random outcomes. While Typma's intentionally bad type system is ridiculous, many accepted programming languages are not as sound as one would think.

References

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

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

[3] 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).

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

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

[6] nLabauthors. *Type Theory*. https://ncatlab.org/nlab/show/type+theory. May 2022.

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

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

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

[10] *SimPPLe*. https://esolangs.org/wiki/SimPPLe. Feb. 2022.

[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).