Typma Programming Language 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 textbook 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 to cause bugs and break 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 with 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 type at compile time. Examples of popular programming languages that have a dynamic type systems are Python and Javascript. Dynamic type systems allow for greater flexibility than static type systems and are popular these days as a result of this flexibility. 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) [2] 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 “undefinedJohn Joe,” making it hard for someone to access there 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 represent as a string, integer, or boolean.

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 avoid 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 apart of a family of languages called esoteric programming languages. The idea of an esoteric programming language is 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) [1]. 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 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 every 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.

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* [5]. IMP features support for arithmetic expression, boolean expressions, variable assignment, if statements, and while statements. Typma adds features like basic functions, concurrency if time allows, and the Typma type system of overly specific type identification and random typing if non is specified. Here are the formal semantics of IMP:
syntax rules
arithmetic expressions:

$$a \in \textbf{Aexp} \; a ::= x|n|a_1 + a_2|a_1 \times a_2$$

boolean expression:

$$b \in \textbf{Bexp}\ b ::= \textbf{true}|\textbf{false}|a_1 < a_2$$

commands:

$$c \in \textbf{Com}\ c ::= \textbf{skip}|a ::= x|c_1; c_2|\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2|$$
$$\textbf{while}\ b\ \textbf{do}\ c$$

evaluation rules
still figuring out how to put them in latex
The Typma programming language project 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 a representation of the code as a tree. This project will use an AST to represent code, abstract syntax tree, 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* [5], 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. For example, the arithmetic expression (2+2)-1 can be evaluated in either big-step or small-step semantics. The big step semantics would look like this:

The small-step method looks like this:

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. This paper will discuss a small sample of previous esoteric languages: Brainfuck, Malbolge, and J.A.V.A..

Brainfuck is a well-known esoteric programming language, with a purpose of having the smallest amount of syntax symbols and the smallest size of compiler of any programming language is described in Morr (2014) [1]. Brainfuck's functionality is based on memory cells each containing an unsigned byte value between 0 and 255. Brainfuck keeps track of a head cell, which is a pointer to a memory cell with the other cells being linearly connected similar to list. Brainfuck has eight characters for syntax: ">" moves the head to the right, "<" moves the head to the left, "+" increments the value of the current cell, "-" decrements the value of the current cell, "," reads an ASCII character from the user and writes the value to the current cell, "." outputs the current cell's value as an ASCII character, "[" skips to the matching closing bracket if the current cell contains a 0, and "]" returns to the matching opening bracket if the current cell does not contain a 0. This is what a Brainfuck hello world program looks like:

```
++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+
[<]<-]>>.>---.+++++++..+++.>>.<-.<.+++.----
--.--------.>>+.>++.
```

Brainfuck is a different kind of esolang compared to Typma. Brainfuck is meant to show the limit of minimal symbols for minimal function, while Typma is a high-level programming language that is just a worse version of language like Python or Javascript. In addition, Brainfuck is not calling any problem of programming languages to attention, on the other hand Typma sets out to mock problems with modern programming languages.

An other famous esoteric programming language is Malbolge, a language that was designed to be near-impossible to both understand and use Morr (2014) [1]. Techniques such as encryption, self-modification, and unpleasant operators combine to make something so difficult to program in, the first program was written two years after Malbolge's specifications were formalized. Going into detail on the specifics of how Malbolge works is out of scope for this paper, but the main idea is that Malbolge is designed to be as difficult to use as possible. Here is a simple cat program in Malbolge:

```
(=BA#9"=<;:3y7x54-21q/p-,+*)"!h\%B0/.
~P<
<:(8&
66#"!~}|{zyxwvu
gJ\%
```

Both Malbolge and Typma serve to frustrate the programmer with features that are designed to frustrate rather than function. However, Malbolge is several leagues above Typma in difficulty of use, with Typma just adding what would be a minor inconvenience compared to the hell that is Malbolge. Malbolge is also a stack based language like Brainfuck where Typma is again a high-level language.

The final esoteric programming language to examine is an obscure language called J.A.V.A., short for Just Another Verbose Annoyance, a language that can be described as an object disoriented language. 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. In addition, J.A.V.A. requires the number of semi-colons to increase Instead of reading a description, looking a helloworld program in J.A.V.A. puts into perspective on how these rules work in the language:

```
ClassFactory factory =
    ClassFactoryProviderSingleton.getSingleton()
   .getFactory(ClassFactoryProvider.CLASS);
factory.setName("Hello, World!");
MethodFactory methods =
    factory.createMethodFactory();;
Method main =
    methods.createMethod("main");;;
StatementBuilder statements =
    main.createStatementBuilder();;;;
Statement statement0 =
    statements.createStatement(0);;;;;
ExpressionBuilder expressions =
    statement0.getExpressionBuilder();;;;;;
Expression helloWorldString =
    expressions.stringLiteral("Hello,
    World!").build();;;;;;;
Expression printHelloWorld =
    expressions.clear().call("println")
   .on(helloWorldString).build();;;;;;;;
statement0.set(printHelloWorld).ignoreResult()
   .save();;;;;;;;;
Statement statement1 =
    statements.createStatement(1);;;;;;;;;;
expressions =
    statement1.getExpressionBuilder();;;;;;;;;;;
Expression return =
    expressions.return().from().method()
   .with(expressions
   .nullLiteral.build());;;;;;;;;;;;
statement1.set(return).ignoreResult()
   .save();;;;;;;;;;;;;
statements.updateStatement(0).set(statement0)
   .save();;;;;;;;;;;;;;
statements.updateStatement(1).set(statement1)
   .save();;;;;;;;;;;;;;;
main.updateStatements.set(statements)
   .save();;;;;;;;;;;;;;;;
factory.addMethod(main);;;;;;;;;;;;;;;;;
factory.build().main();;;;;;;;;;;;;;;;;;
```

Compared to the other esoteric programming languages, J.A.V.A. is the most similar to Typma. They are both high-level languages that serve to mock accepted languages for their problems. The difference is that the problem focused on is completely different. J.A.V.A. is about the problems of Java's class system but Typma is about bad type systems in Python and Javascript.

While many esoteric languages exists, none of them focus on Typma's goal to create an inconvenient type system. None of the esoteric programming languages researched had Typma's feature of requiring extra syntax to specify data types. Many esoteric programming languages have random number generation as a feature of the language but do not apply that to a type system. As a result, the Typma programming language is the first of its kind.

The Typma programming language project takes indirect approach to exposed the problems of a bad type system. On the other hand, TypeDevil takes a direct approach to show the problems of dynamic type systems in the context of Javascript. TypeDevil is a benchmark that generating warnings if inconsistent types are found. TypeDevil is tool that is meant to be used to help real programming project, whereas Typma is not meant to be used for any serious programming project of any kind. Also, TypeDevil is focused on Javascript but Typma related to any language with a faulty dynamic type system.

4 Ethical Considerations

The Typma project is something that can be done in almost any programming language, with the language of choice for this project being OCaml. There is nothing significant about which programming language the Typma parser and interpreter are being written in except there are many design choices and features that do not translate to other languages outside of the functional programming paradigm. This is an accessibility problem because many programmers do not have the experience with functional programming to be able to understand some specific methods of the Typma project or may even have a difficult time reading and understanding the code in general. Programmers have problems when learning how to use a new programming language in general as shown in Shrestha (2020) [3]. Something that can get in the way of learning a new programming language is prior knowledge from another programming language. For example if you are a Java programmer trying to learning Python, a problem that could happen is the habit of placing a semi-colon at the end of every statement will make getting used to Python's lack of semi-colons with this usage more difficult.

The surface problem of learning a new language will happen for anyone trying to understand the code of the Typma project who does not know OCaml. But what makes this an accessibility problem is that Typma's implementation uses features of functional programming. One of these features is declarations of custom types. Type declarations are important to how the different types of statements, like a print statement or a function declaration, are categorized in Typma programs. Typma commands are represented with a type called "cmd" which has different variant depending on what type of expression it is. Here is what the cmd type will be declared as:

```
(* types in typma *)
type typma =
  | int
  | bool
  | str

(* function parameters *)
type param = string * typma option

type cmd =
  | Skip
  | Print of aexp
  | Ass of string * aexp
  | Seq of cmd * cmd
  | If of bexp * cmd * cmd
  | While of bexp * cmd
  | Do of cmd * bexp
  | Fun of param * cmd
```

Languages like Python, Java, and C++ do not have as expressive algebraic data types. Those languages do have objects, but objects are very different from type declarations and trying to understand type declarations as if they were objects defeats the purpose of trying to understand functional programming. Although the programming language being used would not normally affect accessibility to a large enough extent to be noticed, the Typma implementation of functional programming features would make it difficult for someone without a basic understanding of functional programming to understand the methods of the project.

It is important to stress that the Typma programming language is not meant to be used for anything practical, but it is worth it to see how the logical extreme of a bad type system could cause ethical issues if it was ever used. These problems go past making the developer's life harder; if any unspecified data types make it into a released version of software written in Typma, people's lives could hang in the balance of whatever the random number generation decides.

For example, the paper Tachtler (2021) [4] explores ways to ethically make mental health apps with unaccompanied migrant youth in mind. If some version of the Typma programming language was used to make this mental health app, it would be more difficult to make the app function, preventing mental health from being addressed. If the app featured some kind of notification that was time based, like a wellness notification at 9:00 a.m., and the programmer did not specify the type properly to avoid random typing, the notification only has a chance to be played at the right time depending on how the interpreter evaluates the unspecified type. Such a language that allows problems like this to slip by without being noticed could lead to the youth using the app randomly to not see the notification at the right time or not at all, thus failing to help the youth who need help.

A different language would have a type error that would flag this kind of issue. A language like Java requires types to be specified upon declaration or an error will be generated. Typma however will run whether or not the type is specified, leading to situations that programmers do not intend to happen. In a perfect world the programmer would not make the error of forgetting to specify types in accordance to proper Typma conventions, but that is unrealistic to assume that programmers are perfect and if Typma was used these issues would bleed into the real world. Typma is too dangerous to be used because of potential ethical issues it can cause by hurting the basic functionality of apps.

5 Methods

The first component 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 settle the specifics. Although not all of Typma's features have been decided on yet, the language will be a derivative of basic semantics without Typma's type system implemented. Here are the base semantics:

The Typma codebase will consist of seven files: lexer.mll, parser.mly, syntax.ml, env.ml, bigstep.ml, bigStep.ml, smallStep.ml, and main.ml.

The lexer.mll file will contain the specification for the tokens of Typma. Examples of tokens are parenthesis, math symbols, statements like if, and booleans. The parser.mly takes in the tokens and organizes them into the AST of syntax types. The syntax.ml file defines types for math operators and expressions, boolean operators and expressions, comparison operators, and Typma commands. It is possible to write a parser programming that does not use a lexer, but manually parsing code is more difficult and time consuming beacuase reasons. Moreover, the libraries OCamllex and Menhir are set up for a parser and lexer, so comitting 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, there is no need to re-invent the wheel for the parser that will function the same as any other parser.

The env.ml file defines functions for the evaluation of the AST and error types for runtime errors. The bigStep.ml file will have the evaluation of the AST with the big-step version of Typma semantics. The smallStep.ml will have the evaluation of the AST with the small-step version of Typma semantics. The main.ml file will be the file that will be ran in the command line to run a Typma file. An option is given for running Typma in either big or small step semantics depending on the command used to run the Typma file.

Many of the representations of the Typma programming language could be represented will objects as opposed to type declarations. 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 a type declaration. The code for a type declaration 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 type declarations is less work and less code than objects for this project.

While not every situation has been planned out, the general idea of how the Typma programming language functions has been decided. Typma will act exactly like IMP with functions added when proper Typma conventions are followed with type specification. Typma's type specification is for instances of data being used, not on variable declaration. An example of a Typma variable looks like this:

```
a = INT[10];
```

This is an example of the syntax for type specification, but the above code there is no need to specify the type because variable declarations do not store types in Typma. If this variable was printed, the declaration with the INT() specifier would not specify the instance of the variable that is being printed. Take note the Type specifier operator is meant to look at ugly as possible.

```
a = INT[10];

print(a);
```

In order to avoid random typing the use of the variable needs to be specified.

```
a = 10;

print(INT[10]);
```

To recap, variables have no type, only the instances of variables or data has types in Typma.

6 Evaluation

The first part of evaluating the function of the Typma codebase is if it type checks at compile time. This is an indication that the types of the functions return the type that the are supposed to. In addition to type checking, to check if the Type codebase is an accurate representaion 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 to show the codebase is correct. The large number of test cases will prove that the Typma codebase is a proper implementation of the Typma programming language as defined in the formal semantics of Typma.

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

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

If the Typma code base prints a 10, then the function of the code base is shown to be correct.

Some Test cases will have a random number of outcomes, here is an example:

```
print(10);
```

This test case could print out 10, but the random typing could type it as a boolean and print true like a c boolean. These random test cases would have to be ran multiple times to show that the Typma code base is functioning correctly.

If there is enough time, another way to evaluate the Typma code base would be a simple proof that would prove the types are correct for the codebase.

7 Proposed Timeline

Assuming no work in done is the summer, 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 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 stuff to interpreter
- Dec 1: Check with test cases
- Dec 15: Project due

References

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

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

[3] Shrestha, Nischal. “Here We Go Again: Why Is It Difficult for Developers to Learn Another Programming Language?” In: *ICSE ’20* (2020).

[4] Tachtler, Franziska. “Unaccompanied Migrant Youth and Mental Health Technologies: A Social-Ecological Approach to Understanding and Designing”. In: *ACM* (2021).

[5] *The IMP Language*.