Type Systems: Implications in Modern Programming Languages

Author: Sarah Parker

Department of Computer Science, University of Tennessee, Chattanooga

CPSC5100: Theory of Programming Languages

October 7, 2024

Type Systems: Implications in Modern Programming Languages

` Type systems are one of the crucial structures modern programming languages employ to prevent code errors. They are an essential component in maintaining program safety, performance, expressiveness, and reliability. The concept of type systems has existed since the early days of programming; however, there was no explicit need for type systems as assembly and machine code languages acted directly on the hardware. As programming languages evolved and became more abstract, languages like Fortran, COBOL, and LISP created type systems to bring more structure, safety, and clarity to programming ("History of programming," 2024). The different methods and classifications used to define type systems often overlap in their definitions and scopes. This practice results in a system that is unclear and poorly understood. This paper seeks to define the core concepts of type systems more clearly and explore their critical role in modern programming languages.

Type Systems

Overview of Type Systems

Many components are needed to create a compelling and safe programming language. High-level languages seek to make it easy for programmers to write code, understand it, and eventually debug when necessary ("High-level programming" 2024). However, computers do not inherently understand high-level languages. Computers understand machine code instructions called binary, a language consisting only of 1's and 0's. This means a high-level language must somehow be translated effectively into an assembly code (an intermediate code) and then translated once more into machine code (binary). The translation process is completed by a compiler or interpreter, a program that takes a high-level language as input and produces a machine-level code as output ("High-level programming," 2024). A compiler or interpreter uses various methods and rules when translating this code to ensure the syntax (form) and semantics (meaning) are correctly translated (Scott, 2016). Type systems are one of the methods employed by the compiler to ensure program accuracy and prevent type errors. Type

errors occur when the operation is given a different type value than the compiler or interpreter expected (“Type Systems,” 2024).

The term *type* can be defined as the value that describes a piece of data (“Type Systems,” 2024). Primitive types are built into the language, such as integer, character, Boolean, and real. Conversely, the programmer creates a composite type by using a type constructor, such as an array, set, or record (Scott, 2016). Types can be assigned to most expressions (i.e., variables, methods, functions, parameters, etc.) In its most basic form, a type system consists of rules that specify a property type for a construct ("Type System," 2024). Type systems are crucial in providing context and security to codebases. Compilers use type systems to create safe and meaningful code by defining and enforcing constraints on the data. Type systems specify what operations can be performed, the amount of storage needed for a defined type, the range of values stored, inheritance rules, and any interfaces it may implement (“Type Systems in Programming Languages,” 2022). The compiler or interpreter then enforces these rules to ensure that certain operations are only performed on compatible data types. This helps prevent errors during program execution. Each programming language is responsible for creating enforceable rules for its type system either by building it into the interpreter or compiler or by using optional tools (“Type Systems,” 2024). These rules can vary significantly between languages. For example, Java’s type system rules prohibit multiplying an expression containing a string with another expression containing an integer. The string “hello” cannot be multiplied by the integer 5. If this operation were attempted in Java, a type error would be produced during compilation. Python’s type system, however, does not prohibit this type of operation in its rules. If the previous operation were attempted in Python, the results would be “hello hellohellohellohello”.

Evolution of Type Systems in Programming Languages

Fortran, one of the earliest high-level languages dating back to 1958, used a version of implicit typing for its typing system. Variables named with the letters I-N were assigned the type integer. All other variables were assigned the type real ("Fortran," 2024). This allowed for basic error checking but was considerably limited compared to the sophisticated type systems found in modern languages.

As procedural and object-oriented programming was born out of more advanced languages, type systems evolved to manage increasingly complex codes. ALGOL and Pascal, prominent in the 1970s, introduced explicit type declarations, laying the groundwork for future type systems that improved code reliability by providing early error detection through static type-checking (Rizwan, O., 2018). This gave programmers insight into their code's behavior during execution as they could more easily identify it through documentation. In the 1980s and 1990s, advancements in statically typed languages, such as C and ML, further enhanced type safety. Statically typed languages like Java and C# introduced stricter type systems, improving the robustness of code by catching errors at compile-time ("Statically Typed," n.d.).

The rise of dynamic languages like Python and JavaScript in the 2000s emphasized flexibility and ease of use, marking a shift from static to more dynamic type systems (Rizwan, O., 2018). Although these languages gained quick popularity, they also brought new challenges in maintaining type safety. In recent years, there has been a push towards new hybrid languages that combine explicit typing and provide some flexibility from dynamic languages. Rust and TypeScript are two languages that have sought to combine these features and provide a flexible yet safe programming environment ("Type Systems," 2024).

Classifications in Type Systems

Statically and Dynamically Typed

Another way to classify type systems is by determining when type checking occurs. Languages such as C++, Java, and Rust use static type checking, which enforces type checks during compilation.

Type checking ensures that the individual pieces of an expression have appropriate types and are compatible with the operation (Scott, 2016). This allows errors to be caught before the program is run and will throw a type error during compilation. Because types are checked during compilation, there is usually no need to re-check during run time. Therefore, code performs better and executes faster during run time ("Type System," 2024). Statically typed languages, however, tend to be more verbose as they often require explicit type declarations for variables, parameters, and return values.

In contrast, languages like Python, Ruby, and JavaScript use dynamic type checking, which evaluates and assigns types during run time and only if that piece of code is executed (“Type System,” 2024). Dynamically typed languages provide the programmer with greater flexibility and ease of use (“Type Systems in Programming Languages,” 2022). However, some noted difficulties with dynamically typed languages are slower performance, increased risk of run-time errors, difficulty refactoring code, and long-term complexities in maintaining the code (“Introduction to Type Systems,” 2022). To help alleviate these issues, some dynamically typed languages have introduced static analysis tools. For example, Python offers mypy, an optional static type checker that allows programmers to annotate their code with type hints (“mypy,” 2014). TypeScript, a superset of JavaScript, is another example of a dynamically typed language that uses optional static typing. These languages and tools provide programmers who want both flexibility and type safety with solid middle-ground options that combine the benefits of both static and dynamic type checking.

Implicitly versus Explicitly Typed

A fundamental distinction in type systems is whether they are explicitly typed or implicitly typed. Explicitly typed means that the programmer must declare the type when writing code. Implicitly typed refers to the method by which the compiler or interpreter will evaluate and assign a type without the declaration of types from the programmer (“Type System,” 2024). One can argue that explicit type

systems should only be evaluated in the context of a statically typed language, as dynamic languages always use a form of type inference and implicit type assignment (Pierce, 2002). In more explicitly typed languages, such as Javas, C++, and Rust, the programmer must plainly state the types of variables, function parameters, and return values. As languages have evolved, most are not strictly explicitly or implicitly typed. Instead, most modern languages use a combination of both. Until recently, Java only allowed explicit typing. However, with the release of Java 10, some forms of implicit typing and inference are now available ("Var Keyword," 2023). The following two expressions will further describe these characteristics. The following example, written in Java, is an explicitly typed expression because the variable type is declared before the name: String city = "Chattanooga";. The following is an example of an implicitly typed variable in Java: var city = "Chattanooga";. The Java compiler will assign a type based on the surrounding context. The type is not explicitly stated; instead, the abstract "var" is used. The type is implicitly assigned during compile time when the compiler evaluates this expression. Java has several rules for when using the var keyword is allowed. It can only be used in local variables, it cannot be declared without being initialized simultaneously, and it cannot be used in parameters and return functions ("Var Keyword," 2023). One could argue whether this is truly a form of implicit typing as the programmer must still declare the keyword var to use it. However, given that the compiler infers the type it does make sense to categorize it as an implicit expression. Now that the variable's type has been established, if the program attempts to misuse it, such as adding it to an int variable, the compiler will flag this as an error before the code is executed. This often results in faster performance during runtime because type correctness is ensured beforehand.

In contrast, dynamically typed languages such as Python or JavaScript allow the programmer to omit type declarations altogether. These languages rely on the compiler or interpreter to infer the type based on context. Referring to our above example, in Python, the same expression could be written as: city = "Chattanooga";. Python will then infer and assign a type during run time as it is a dynamically

typed language. Another important feature of Python is that it reserves the right to change the type depending on the operation used. For example, the expression num = 5 could be reduced to a type integer or a String. If the specified operation is a mathematical computation, the variable num would likely be implicitly assigned as an integer. However, if a type string was needed in a different operation, it could also be implicitly converted to a string. Implicit typing is more flexible and more accessible for the programmer to write. The tradeoff for this flexibility is less optimized code, more run-time type errors, and immense difficulty locating type errors when they occur (Clark, 2018).

Type Inference versus Duck Typing

Two more methods related to implicit and explicit typing are type inference and duck typing. Type inference is a method used in both statically and dynamically typed languages. It allows the programmer to omit the type in a variable declaration (implicitly typed) and instead relies on the compiler to fill in the missing types. Type inference is the backbone for implicitly typed languages as it provides the structure to evaluate the expression in the context of the surrounding code and assign the relevant type (Scott, 2016). ML and Haskell are the two most notable languages in which type inference is prominently and more complexly used types (Duggan & Bent, 1996). In these languages, programmers can declare an object type. In languages like Java, objects cannot have a declared type, but objects can be cast to a type to use in operations. Type inference is not limited to implicitly typed languages. Languages that are explicitly typed use type inferences for instantiating generic type variables using polymorphic functions (Scott, 2016).

Another related method is called duck typing. This concept originates from the saying, “If it looks like a duck and walks like a duck, then it must be a duck” ("Duck Typing," 2024). This is essential in object-oriented languages. Duck typing focuses more on the object’s behavior rather than figuring out exactly what the type is. Objects are identified as compatible and equivalent if all their methods and

properties are the same (“Duck Typing,” 2024). Therefore, the actual type does not matter if the objects can be treated as equivalent. The following Python code adapted from (“Duck Typing,” 2024) provides a clear example of duck typing.

```
class Duck:
    def swim(self):
        print(“I am a duck, I can swim”)

    def fly(self):
        print(“I am a duck, I can fly!”)

class Dog:
    def swim(self):
        print(“I am a dog, I can swim”)

for animal in [Duck(), Dog()]:
        animal.swim()
        animal.fly()
```

Output:

```
I am a duck, I can swim
I am a duck, I can fly
I am a dog, I can swim
AttributeError: ‘Dog’ object has no attribute ‘fly’
```

The dog could be considered a duck object because they both have the swim function. However, once the dog object does not have a fly function, it can no longer be considered a duck, and an attribute error is produced.

Type Safety

Type safety can be defined as the “extent to which a programming language discourages or prevents type errors” (“Type Safety,” 2024). A combination of methods attempts to describe and define type safety. A type-safe language prohibits operations that could lead to dangerous behavior, such as trying to access memory that has not been allocated or attempting to modify fixed data. In the early 1990s, Andrew Wright and Matthias Felleisen established a commonly accepted standard for evaluating a language’s safety. They purported that a type-safe language must have progress and preservation (“Type Safety”, 2024). They further defined progress as a program that can never get stuck; every expression is either assigned a value, or a value can be assigned through a logical reduction. Type safety prohibits operations on irrelevant data. Preservation is defined as each expression keeping the same type after every execution step (Type Safety, 2024). Programmers must be aware of language vulnerabilities to help prevent memory errors and invalid type operations and improve code stability and reliability.

Strongly Versus Weakly Typed

Historically, languages have often been categorized as strongly or weakly typed in reference to their type safety. However, multiple authors agree that this is a poor generalization when describing type safety (“Strong and Weak Typing,” 2024). Regardless, these terms are still used frequently, and an attempt to describe them is warranted.

Generally, a strongly typed language detects errors, ensures correctness, and prohibits objects from acting in a way not defined for that type (Scott, 2016). It is said that a strongly typed program cannot “go wrong” (“Type Safety,” 2024). Although this is a vast understatement for strongly typed languages, the general meaning is well understood. A strongly typed program will not behave unexpectedly when following its system’s syntactic typing rules (“Strong and Weak Typing,” 2024). Java,

Haskell, and Rust are considered strongly typed languages and are designed with strict type systems that catch errors early in development due to their static type checking (“Strong and Weak Typing, 2024). Casting, a method used to convert data types from one type to another, is also used to define type safety (“Type Casting,” 2024). Implicit casting is when the compiler changes the type automatically to fit within the rules of the designated operation. Explicit casting is when the programmer assigns a different type to the specified expression. Explicit casting is felt to be a characteristic of a more strongly typed language as it prevents undesired conversions of types (“Strong and Weak Typing,” 2024). Implicit casting is considered a characteristic of a weakly typed language. Java has precise rules regarding casting and conversions. The Java programmer needs to state the newly desired type explicitly. There are two common types of casting for primitive data types in Java: widening and narrowing (“Java Type Casting,” n.d.). In widening, conversions occur implicitly because the data type goes from a smaller to a larger memory space. For example, the variable int x = 2; could be cast to a double by specifying double y = x. The result would be a variable containing the double 2.0. The opposite is not valid. A double cannot be implicitly converted to an int. Conversions between variables that go from a larger memory space (i.e., double) to a smaller memory space (i.e., int) are called narrowing type casting (“Java Type Casting,” n.d.). This cannot be implicitly converted because a loss of data would occur. Java requires the programmer to acknowledge this by explicitly stating the type when this type of conversion is desired. Using our example from above, double x = 2.3 could be cast to an int by specifying int y = (int)x. The result would be y = 2, and the .3 would be dropped. Explicit casting notifies the compiler that the programmer is aware of a data loss and that it is ok to proceed with the operation (“Java Type Casting,” n.d.). Another restriction on type conversions in Java is related to strings. Primitive data types (integers [ints], double, float, etc.) In Java, programmers can use several built-in methods to convert a number to a string, such as toString() or valueOf(). On the contrary, to convert a string to an int, the method parseInt() can scan a string for an integer and then save it as an int (“Java Type Casting,” n.d.). Rust, a modern language, may

be considered even more strongly typed than Java as it does not allow any implicit conversions (“Type Conversion,” 2024).

Conversely, a weakly typed language often allows the implicit casting of one value type to a different type or allows pointers to be used as numeric values in computations ("Strong and Weak Typing," 2024). JavaScript, a weakly typed language, allows the concatenation of a string and integer type. Combining the “Hello World” string with a ‘+’ operator and the integer 2024 would produce the following output: “Hello World 2024”. JavaScript automatically casts the integer variable 2024 to a string type (Ikusika, 2022). Therefore, the original operation is converted so the program can effectively concatenate two strings together. Implicit casting alters the value’s type to execute the desired expression effectively. Although this may seem more straightforward and shorter to code, it can lead to unexpected values and confusing output. A type error would result from attempting this operation in a strongly typed language such as Python because there is no defined rule for adding an integer type and a string type together. Python does not allow the implicit conversion of types (Mathur, 2024).

Memory Safety

Memory safety is essential when discussing type safety, especially in languages that allow direct memory management, such as C and C++. Type systems can use the compiler to check for dangling pointers, out-of-bounds accesses, and buffer overflows (“Type Safety,” 2024). This can address security vulnerabilities and prevent program crashes during runtime. Dangling pointers are a type of memory breach that results in a pointer referencing an invalid memory address ("Dangling Pointer," 2024). Accessing and attempting to write to an invalid memory address can cause unexpected or fatal errors for the program. Few languages can claim to be genuinely type-safe, as doing so would require manually checking thousands of cases. Languages like Rust and Standard ML are considered highly type-safe due to their strict type-checking rules and memory safety guarantees (“Type Safety,” 2024). Rust uses a

system of ownership and borrowing to ensure that memory errors, such as null pointer dereferencing and buffer overflows, are eliminated at compile time (Yegulalp, S., 2024). This is one of the primary reasons Rust has gained such popularity within the last few years as a highly safe yet fast and easy-to-learn language. Haskell is a language believed to be type-safe as long as certain “escape” features are not utilized (“Type Safety, 2024).

Implications for the Classification of Type Systems

Although the different classifications and definitions of type systems will likely continue to spark debates among professionals in programming, all can agree that type systems play a crucial role in the development process. There are tradeoffs to each classification system discussed, and the programmer needs to have a solid understanding of the pros and cons of different languages when developing software to utilize its features best and be aware of vulnerabilities. While explicit typing can be more cumbersome and time-consuming for the programmer, it allows more transparent documentation and direction in the codebase. Statically checked programs allow type errors to be caught at compile time, making it ideal to catch bugs before executing the code. Dynamically checked languages may offer the advantage of implicit typing and code that is directly processed and executed. However, this can also lead to a greater risk of run-time errors. Refactoring and code maintenance can also be more difficult due to the lack of documentation that comes with implicit typing. While implicit typing can allow greater flexibility for the programmer, it can also introduce unwanted behavior in programs.

Future of Type Systems

Hybrid Model

As languages have evolved, programmers have seen more robust systems that blend static and dynamic typing (OpenAI, 2024). These languages have often been described as having hybrid or blended type systems. Their popularity has increased as they allow programmers to choose the right level of type

enforcement for different parts of their applications. Several modern languages have adopted these structures. TypeScript introduced syntax and static type checks that can be added to JavaScript code to check for safety and efficiency during compile time while still allowing the flexibility of dynamic typing for the rest of the program (“TypeScript,” (n.d)). Swift, a popular language for iOS and macOS app development, is also statically typed but takes advantage of type inference and optional types to write safe, flexible code (OpenAI, 2024). Clojure is a dynamically typed variation of the language Lisp. It has added support for optional statically typed expressions using the core.typed library (OpenAI, 2024). The future of type systems in modern programming languages will likely emphasize greater flexibility, safety, and expressiveness and continue to combine elements of previously accepted type systems.

Advanced Features

Type systems will likely continue to produce more advanced features like polymorphic functions, gradual typing, and dependent subtypes, which will become more common while ensuring vigorous compile-time type checks (Scott, 2016). Parametric polymorphism is an advanced feature of type systems that has garnered much attention over recent years. The principal component of this feature is the ability of a language to assign different types of value dependent on what the operation requires. This can be done implicitly or explicitly. Scott (2016) defined explicit parametric polymorphism, or generics, as a component that “allows the programmer to specify type parameters when declaring a subroutine or class.” This shift will likely include more powerful type-level programming, allowing types to encode more complex constraints and logic. Increased support for generic and polymorphic types will enable code reuse without sacrificing type safety (Scott, 2016).

Gradual typing is a feature that has gained immense popularity as it combines static and dynamic type checking. These languages allow programmers to optionally state types, which are then checked during compile time. Variables left without a type are assigned the type dynamic as a signal to

the compiler that they will be type-checked during runtime (“Gradual Typing,” 2024). These types of gradual typing languages were derived from dynamically checked languages. For example, languages that have evolved from this model are TypeScript from JavaScript and Hack for PHP (“Gradual Typing,” 2024).

Subtyping is another advanced feature and is also considered a form of polymorphism. If an element is a subtype of a superset, then any function designed to work with the superset will work with the subtype as type compatibility is ensured (“Subtyping,” 2024). This is especially helpful when designing interfaces to allow different aspects of a program to work together cohesively while also allowing code reusability. If a function was comparing two numbers, then an integer and a float could be used as they are both subtypes of numbers.

A key area of interest in programming is AI (artificial intelligence), and type systems provide vital structure and reliability to this field. Walker (n.d.), the CEO of KLU AI, stated, “In natural language processing (NLP), they manage and validate linguistic data structures like syntax trees and semantic representations. Type systems ensure the accuracy and coherence of language processing tasks such as sentiment analysis and text classification.” AI is dependent on type systems not only for processing and reliability but also for standardization and optimization. The future of type systems will undoubtedly continue to advance and start blending features rapidly to keep up with the needs of independent developers, machine learning, and AI.

Conclusion

Type systems continue to evolve along with advancements in programming languages. As modern applications become increasingly complex, the need for reliable, efficient, and type-safe code has never been more critical. The rise of functional programming, type inference, and hybrid typing systems, such as those in TypeScript, demonstrates how type systems adapt to developers' needs and

the demands of modern software. The development of machine learning and artificial intelligence type systems may offer better support to help programmers manage complexity while improving accuracy and performance. The debate between static and dynamic type checking, strong and weak typing, and explicit and implicit typing will likely persist as different cases require different trade-offs. As languages evolve, however, we can expect type systems to become even more integral to creating safe, efficient, and maintainable code.

References

Clark, D. P. (2018, November 6). *What is the Difference Between Implicit vs. Explicit Programming?* CloudBees. Retrieved September 24, 2024, from https://www.cloudbees.com/blog/what-is-the-difference-between-implicit-vs-explicit-programming

Dangling pointer. (2024, September 1). In *Wikipedia*. https://en.wikipedia.org/wiki/Dangling_pointer

Duck typing. (2024, August 13). In *Wikipedia*. https://en.wikipedia.org/wiki/Duck_typing

Duggan, D., & Bent, F. (1996). Explaining type inference. *Science of Computer Programming*, *27*(1), 37-83. https://doi.org/10.1016/0167-6423(95)00007-0

Fortran. (2024, September 4). In *Wikipedia*. https://en.wikipedia.org/wiki/Fortran

Gradual typing. (2024, July 29). In *Wikipedia*. https://en.wikipedia.org/wiki/Gradual_typing

High-level programming language. (2024, September 2). In *Wikipedia*. https://en.wikipedia.org/wiki/High-level_programming_language

History of programming languages. (2024, September 12). In *Wikipedia*. https://en.wikipedia.org/wiki/History_of_programming_languages

Ikusika, I. (2022, October 9). *Type safety and strong typing—What are these terms?* DEV. Retrieved September 14, 2024, from https://dev.to/ayodejii/type-safety-and-strong-typing-what-are-these-terms-3io5

Introduction to type systems. (2024, June 5). In *Ada Beat* https://adabeat.com/fp/introduction-to-type-systems/

Java type casting. (n.d.). Programiz. Retrieved September 20, 2024, from https://www.programiz.com/java-programming/typecasting

Mathur, T. (2024, April 7). *Implicit Type Conversions in Python*. Scaler Topics. Retrieved September 24, 2024, from https://www.scaler.com/topics/python/implicit-type-conversion-in-python/

Mypy. (2014). mypy. Retrieved September 23, 2024, from https://mypy-lang.org/

OpenAI. (2024). *ChatGPT* [Large language model]. https://chatgpt.com

Pierce, B. (2002). *Types and Programming Languages* (1st ed., pp. 17–21). MIT Press.

Scott, M. L. (2016). *Programming Language Pragmatics*(4th ed., pp. 491–537). Elsevier.

Statically typed. (n.d.). Amazing Algorithms. Retrieved September 23, 2024, from https://amazingalgorithms.com/definitions/statically-typed/

Strong and weak typing. (2024, May 13). In *Wikipedia*. https://en.wikipedia.org/wiki/Strong_and_weak_typing

Subtyping. (2024, March 29). In *Wikipedia*. https://en.wikipedia.org/wiki/Subtyping

TypeScript. (n.d.). TypeScript. Retrieved September 24, 2024, from https://www.typescriptlang.org/

Type conversion. (2024, September 23). In *Wikipedia*. https://en.wikipedia.org/wiki/Type_conversion#

Type safety. (2024, July 8). In *Wikipedia*. https://en.wikipedia.org/wiki/Type_safety

Type system. (2024, August 29). In *Wikipedia*. https://en.wikipedia.org/wiki/Type_system#Static_typing

Type Systems In Programming Languages (2022, July 31). *In TekTutorialsHub.*

https://www.tektutorialshub.com/programming/type-systems-in-programming-languages/

Var Keyword in Java (2023, April 22). Geeks for Geeks. Retrieved September 24, 2024, from https://www.geeksforgeeks.org/var-keyword-in-java/

Walker, S. M., II (n.d.). *What is a type system*. KLU. Retrieved September 25, 2024, from https://klu.ai/glossary/type-system

Yegulalp, S. (2024, April 3). *Rust memory safety explained*. InfoWorld. https://www.infoworld.com/article/2336661/rust-memory-safety-explained.html