A survey of type systems in programming languages

Óbuda University
John von Neumann Faculty of Informatics

András Sallai

2021

Contents

Bevezető

Statikus nyelvekkel először az egyetemi tanulmányaim során találkoztam C# és Java tárgyakon. Ekkor már volt pár év programozói tapasztalatom, de csak dinamikus nyelvekkel. Kezdetben rendkívül körülményesnek éreztem a munkát C#-ban vagy Java-ban. A kód terjengősnek tűnt, a programok nem működtek, amíg látszólag minden a helyén nem volt, apró változtatások is hibák egész áradatát hozták magukkal, amiket mind ki kellett javítanom, mielőtt újra futtathattam volna a kódomat. Nem tartott ám sokáig, mire elkezdtem érezni az előnyeiket. Miután kezdtem megszokni őket, ha újra elővettem munkámhoz a megszokott dinamikus nyelveket, úgy éreztem, visszafelé teszek egy lépést. Folyamatos próbálgatás, változtatás majd újra próbálgatás, bizonytalanság a program helyességét illetően. A statikus nyelvek segítségével hatékonyabban és magabiztosabban dolgoztam. A type checker segítségével jobban átgondoltam a megoldásaimat, magabiztosságot adott a változtatásokhoz. Egyre jobban érdekelt a statikus típusellenőrzés és a típusrendszerek témaköre. Kíváncsi lettem, mit mutathatnak még fel programnyelvek, amivel segíthetik a programozó munkáját és növelhetik a szoftverek minőségét, ez a kíváncsiság vezetett ahhoz, hogy a típusrendszereket válasszam szakdolgozatom témájaként. A célom e munkával, hogy mélyebb megértést szerezzek a programnyelvek működéséről - statikus és dinamikus nyelvekről egyaránt. Először áttekintem a típusrendszerek elméleti hátterét és történetét. Megvizsgálom, hogy a típuselmélet elemei hogyan formálták az ismert programnyelveket. Bemutatom kódrészleteken keresztül, hogy bizonyos típusrendszer elemek hogyan segíthetik a programozót munkájában. Összehasonlítom ismert nyelvek típusrendszereit és megpróbálok egy áttekintést adni a típuselmélet újabb keletű gyakorlati vívmányairól.

Introduction

My first real exposure to statically typed languages was at the university with C# and Java. I already had a few years of programming experience but only with dynamically typed languages. At first, C# and Java felt cumbersome. The code seemed too verbose, programs didn't work until everything was right, even a small change triggered a whole chain of errors and the whole thing had to be fixed before I could run my code again. But it didn't take long until I started seeing their benefits. After spending some time with statically typed languages, each time I went back to the dynamic ones it felt like a step backwards. Continuous trial and error, edit-run-debug cycles, uncertainty about the correctness of my code. Static languages helped me write code more effectively and more confidently. The type checker made me to think more thoroughly, it gave me confidence when changing parts of the program. I became more and more interested in the possibilities of static checking and type systems. I wondered what features can programming languages provide to increase programmer productivity and improve software quality, this curiosity led me to choosing type systems as the topic of this work. My goal with this thesis is to gain a more fundamental understanding of the semantics of programming languages - both static and dynamic. First, I'll survey the history and theoretical background of type systems. I'll look at how type systems concepts shaped programming languages. I'll demonstrate, through example programs the benefits of certain type systems features. I'll compare the type systems of widely used programming languages and will try to give an overview of the possibilities of recent advances in type systems and programming language design.

1 Type systems

> Modern software engineering recognizes a broad range of formal methods for helping ensure that a system behaves correctly […] by far the most popular and best established lightweight formal methods are type systems. [38:1]

What is a “formal method”?

> Formal methods are a particular kind of mathematically rigorous techniques for the specification, development and verification of software and hardware systems. The use of formal methods for software and hardware design is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analysis can contribute to the reliability and robustness of a design. [52]

In contrast, an empirical method - that is based on experimental results - for verifying software systems would be testing.

One of many formal methods, a type system is a set of rules that associate a property called a type to various constructs in a computer program. A type defines a range of values as well as possible operations on instances of that type. In a hardware or compiler context, a type defines the size and memory layout of values of that type. Here I will focus on types from the perspective of type checking.

> A type is simply a property with which a program is implicitly or explicitly annotated before runtime. Type declarations are invariants that hold for all executions of a program, and can be expressed as statements such as “this variable always holds a String object,” or “this function always returns a tree expression.” [44:100]

The main purpose of type systems is to reduce possibilities for bugs in computer programs. The type system associates a type with each value in the program and then by examining the flow of these values attempts to prove that no operation violates its rules.

Such a violation is called a type error. It is an inconsistency in a program according to the type system's rules. Exactly what constitutes a type error is defined by the type system of the language. A program that violates the rules of its type system is often called "ill typed". A program that conforms to those rules is a "well typed" program. [53]

> Type systems are used to define the notion of well typing, which is itself a static approximation of good behavior […] Well typing further facilitates program development by trapping execution errors before run time. [8:6]

There are languages that don't associate a type with values, these are called untyped languages. Most assembly languages fall into this category. In these languages, invalid operations might be applied to values which could result in a fixed value, a fault, an exception or an unspecified effect.

Typed languages can enforce good behavior by performing static (without executing the program) checks to prevent ill typed programs from ever running. Untyped languages can enforce good behavior by performing sufficiently detailed runtime checks to rule out errors. For example, they may check array bounds or division operations and generate recoverable exceptions. This checking process during runtime is called dynamic checking. [8]

By using the facilities provided by the type system, we can add more information in our programs. We can create a safety-net against execution errors by making it possible for automated tools to verify the steps we take throughout our code. Through types, we can add more meaning, a deeper, more solid structure. That structure aids us when we later change parts of the program or add new components to it.

The way I see it, type systems are the glue between mathematical logic and computer programs. They make it possible to prove that our programs behave correctly. Advanced, sophisticated type systems allow us to express a finer, more precise structure and thus more properties of our software may be proven by automated tools and more optimization may be carried out to improve performance.

1.1 Type theory

Type theory is a branch of mathematical symbolic logic: a system of representing logical expressions through the use of symbols. It was conceived in the beginning of the 20th century by Bertrand Russell in order to resolve contradictions present in set theory. Type theories (there are many, like Alonzo Church's Simply Typed Lambda calculus or Per Martin-Löf's Intuitionistic Type Theory) are formal systems which means they define rules for inferring theorems (statements) from axioms. [48]

> Type theory is a formal language, essentially a set of rules for rewriting certain strings of symbols, that describes the introduction of types and their terms, and computations with these, in a sensible way. [65]

Type theory lays down the theoretical foundation for the type systems found in programming languages and the type checking algorithms behind them.

1.2 Formalization of type systems

> How can we guarantee that well typed programs are really well behaved? […] Formal type systems are the mathematical characterizations of the informal type systems that are described in programming language manuals. Once a type system is formalized, we can attempt to prove a type soundness theorem stating that well typed programs are well behaved. If such a soundness theorem holds, we say that the type system is sound. [8:7]

Most materials on the formalization of type systems are dense and get very abstract quickly. Here, I'd like to briefly introduce the basic concepts and standard notation used when discussing formal type systems.

1.2.1 Judgements and rules

An expression is a syntactically correct fragment of a program that can be evaluated to a value. Type systems associate expressions with types. We call this the *has type* relationship: $e : M$, where the expression e has type M. This is called a typing judgement or just judgement. [19]

Judgements are used to build inference rules of the form

$$\frac{J_1 \ldots J_n}{J}$$

Where the judgements $J_1 \ldots J_n$ above the line are called the premisses and J is the conclusion. We read the above expression as "from the premisses $J_1 \ldots J_n$ we can conclude J". If there are no premisses (meaning if n is 0), then the rule is an axiom. An inference rule can be read as stating that the premises are sufficient for the conclusion: to show J, it is enough to show $J_1 \ldots J_n$. The collection of such typing rules form the type system of a programming language.

> Type rules assert the validity of certain judgments on the basis of other judgments that are already known to be valid. The process gets off the ground by some intrinsically valid judgment [8:10]

In a program, the type of a variable can only be decided by looking at its context (or typing environment) which is a finite sequence of bindings of program variables to types and is defined by the declarations of the variables. We can think of context as a lookup table of (variable, type) pairs:

$$x_1 : A_1, \ldots \; x_n : A_n$$

> The role of a type system is to impose constraints on the formations of phrases that are sensitive to the context in which they occur. [20:36]

In the standard notation, context is denoted by the greek letter Gamma:

$$\Gamma \vdash e : T$$

Which we read as "expression e has type T in context Γ".

1.2.2 Derivations

> A derivation of a judgment is a finite composition of rules, starting with axioms and ending with that judgment. It can be thought of as a tree in which each node is a rule whose children are derivations of its premises. We sometimes say that a derivation of J is evidence for the validity of an inductively defined judgment J. [20:15]

If we can construct a derivation for a judgement then we can say that the judgement is valid.

Once the inference rules are constructed, a type checking algorithm can take a program and through constructing derivations, decide if the program is well typed or not. If a program fragment violates the rules of the type system then we say it has a typing error. Most languages report a "Type Error" in such cases.

> A term M is well typed for an environment Γ, if there is a type A such that $\Gamma \vdash M : A$ is a valid judgment; that is, if the term M can be given some type. [8:11] The discovery of a derivation (and hence of a type) for a term is called the type inference problem.

Our aim when constructing a derivation is to infer the type(s) of an expression. The typing rules in the judgements that make up the derivation serve as markers for assigning types to the expression at the root of the derivation. [20], [8], [39]

1.3 A short history of type systems

The origins of type systems and type theory go back to the early 1900s. Type systems were first formalized as a means to avoid Russell's paradox [54]. It was a mathemat-

ical framework at first, "a field of study in logic, mathematics, and philosophy" as Pierce [38] puts it, with no connection to programming or computers.

> During the twentieth century, types have become standard tools in logic, especially in proof theory […] and have permeated the language of philosophy and science. [38:1]

The first type systems in programming language context appeared in the 1950s, when the designers of the Fortran (Backus) and Algol-60 (Naur et al.) languages wanted to make numerical computations more efficient by distinguishing between integer-valued arithmetic expressions and real-valued ones. This allowed the compiler to generate the appropriate machine instruction making the program more efficient. [38], [13]

In the 1960s, the Curry-Howard correspondence was discovered by Haskell Curry and William Alvin Howard. It is the direct relationship between computer programs and mathematical proofs, the link between logic and computation. [55]

The Pascal programming language with "strong typing" was developed in the 1970s (Wirth), so was Martin Löf's Type Theory born and the ML family of languages created.

> In the late 1950s and early 1960s, this classification was extended to structured data (arrays of records, etc.) and higher-order functions. In the 1970s, a number of even richer concepts (parametric polymorphism, abstract data types, module systems, and subtyping) were introduced, and type systems emerged as a field in its own right. At the same time, computer scientists began to be aware of the connections between the type systems found in programming languages and those studied in mathematical logic, leading to a rich interplay that continues to the present. [38:10]

The 1980s brought existential types, dependent types and effect systems. Object calculus (an attempt at clarifying the fundamental features of object oriented languages) and typed intermediate languages (a way of preserving type information through compilation, possibly even in the final target language) have been pioneered in the 1990s

by Cardelli [2] and Monnier [4] respectively. Most of these topics, unfortunately, remain hidden within academic papers and research projects.

2 Language/type safety

> A safe language is one that protects its own abstractions […] Every high-level language provides abstractions of machine services. Safety refers to the language's ability to guarantee the integrity of these abstractions and of higher-level abstractions introduced by the programmer using the definitional facilities of the language. [38:6]

As [38] puts it, the abstraction of a safe language can be used "abstractly", whereas in an unsafe language it is necessary to keep in mind the low level details, like how data is structured in memory or how allocations take place in order to understand how the program might misbehave.

Note that "language safety" can be achieved by static type checking but also by run-time checks, like array-bounds checking which is performed by many languages during runtime.

Chappell [9] says that a programming language or language construct is type-safe if it forbids operations that are incorrect for the types on which they operate. The author notes that some languages may discourage incorrect operations or make them difficult without completely forbidding them.

Cardelli [8] differentiates between trapped errors, that cause execution to stop immediately and untrapped errors that go unnoticed and later cause arbitrary behaviour. An untrapped error, for example, is accessing data past the end of an array in absence of run time bounds checks. A trapped error would be division by zero or accessing an illegal address. He calls a language safe if untrapped errors are impossible in it. The author suggests declaring a subset of possible execution errors as forbidden errors (all of the untrapped and some of the trapped errors) and says that a program can be called "well behaved" if no such forbidden errors can happen during execution.

> Most programming languages exhibit a phase distinction between the static and dynamic phases of processing. The static phase consists of parsing and type checking to ensure that the program is well-formed; the dynamic phase consists of execution of well-formed programs. A language is said to be safe exactly when well-formed programs are well-behaved when executed. [20:35]

To summarize the thoughts of the authors above: a language can be called safe if every program written in it that can reach execution is well behaved. A well behaved program is one that does not produce forbidden / untrapped errors. In a theoretical context, "type soundness" is usually used instead of safety but the two are more or less synonyms.

In a more practical sense, soundness is the ability for a type checker to catch every single error that might happen at runtime. This sometimes means catching errors that will not actually happen at run time. As a side note "completeness" in type systems means the "inverse" of soundness: it is the ability for a type checker to only ever catch errors that would happen at runtime. This comes at the cost of sometimes missing errors that will happen at runtime. [66]

> In modern languages, type systems are sound (they prevent what they claim to) but not complete (they reject programs they need not reject). Soundness is important because it lets language users and language implementers rely on X never happening […] Type systems are not complete because for almost anything you might like to check statically, it is impossible to implement a static checker that given any program in your language (a) always terminates, (b) is sound, and (c) is complete. [12:15]

Regardless of whether type checking happens statically (before running the program) or dynamically (while the program is running), a language may or may not be called safe. For example, there might be operations that would pass type checking but produce an error during runtime, like division by zero. Enhancing a type system so that it could protect against division by zero errors would make it too restrictive (too many programs would be ruled out as ill-formed). It is not possible to predict statically that

an expression would evaluate to zero, so if we want our language to be safe, we need to add dynamic (runtime) checks. Even though it is not part of the static type system, such a language is still considered safe. [20]

2.1 Formal soundness: preservation and progress

Formally, a language can be called "sound" or "type safe" if the following properties hold [20]:

1. If $e : \tau$ and $e \to e'$, then $e' : \tau$
2. If $e : \tau$, then either e is a value or there exists e' such that $e \to e'$

The first property is called "preservation": it states that evaluation preserves the type of an expression. The second property is called "progress", it states that a well typed expression (one that passes the type checker) is either a value or it can be further evaluated.

> Progress says that if a term passes the type-checker, it will be able to make a step of evaluation (unless it is already a value); preservation says that the result of this step will have the same type as the original. If we interleave these steps (first progress, then preservation; repeat), we can conclude that the final answer will indeed have the same type as the original, so the type system is indeed sound. [28:146]

Together they state that if we take an expression (or a piece of program) which has type t and evaluate it then the resulting value will also have type t. [28]

Bonnaire-Sergeant [67] summarizes "preservation" beautifully:

> The essential property of type soundness is "type preservation", which says that a type system's approximation of the result of a computation (a type) is "preserved" at every intermediate computational stage. In other words, the type system's job is to model a computation, and this model should not lose accuracy as the program evaluates.

2.2 Should languages be safe?

Safety reduces debugging time by adding fail-stop behavior in case of execution errors. Many security problems exist because of buffer overflows made possible by unsafe casting and pointer arithmetic operations. Languages that provide safety through bounds checking provide protection against such sources of exploits. Safety guarantees the integrity of run time structures, and therefore enables garbage collection. [8] Yet there are still unsafe languages in widespread use, mainly for one reason: performance.

> Some languages, like C, are deliberately unsafe because of performance considerations: the run time checks needed to achieve safety are sometimes considered too expensive. […] Thus, the choice between a safe and unsafe language may be ultimately related to a trade- off between development and maintenance time, and execution time. [8:5]

3 Type checking

Type checking is the process of deciding whether a term is well typed or not. A type checker verifies that the constraints posed by the type system are not violated by the program. Type checking can be done by automated tools called type checkers, which are usually built into compilers or linkers. [43]

There are two main branches of languages with regards to type checking (or "typing"): static and dynamic. Statically typed languages, where variables have a "static" or "unchangeable" type, carry out type checking before the program is actually run. Dynamically typed languages do type checking during run-time, since variables may change their types during program execution. Both sides have their advantages and disadvantages.

> The debate regarding the advantages and drawbacks of static or dynamic type systems is ongoing in both academia and the software industry. While statically typed programming languages such as C, C++, and Java

> dominated the software market for many years, dynamically typed programming languages such as Ruby or JavaScript are increasingly gaining ground—especially in web development. The fact that the debate is still lively is not surprising, because settling it demands the presence of a theory of the respective advantages and disadvantages of static and dynamic typing, supported by empirical evidence. Unfortunately, such evidence is still lacking. [18]

Proponents of dynamic languages criticize static ones as being too rigid and cumbersome to work with. The other camp complains that dynamic languages offer little protection against logical errors and let too many of the errors happen at runtime. I'll discuss each approach separately in a later section.

Terms like "strongly typed" and "weakly typed" are also in wide use but their meaning is not clearly defined and are often used to mean certain combinations of language attributes. They are used to refer to how much implicit type conversion (also known as "coercion") is done by the language, or if the it prevents memory-safety bugs (array bounds checking or use-after-free) or if type checking is done statically or dynamically. It is best to avoid these ambiguous terms.

3.1 Type checking algorithms

The general idea behind type checking algorithms is the following:

1. Parse the source code into an "Abstract Syntax Tree" (AST)
2. Annotate each node in the AST with a "type annotation"
3. Generate constraints: enumerate all the known relations between the types of the nodes.
4. Use constraint solving to see if the constraints can be satisfied.

3.2 Static type checking

Programmers make errors. Advanced programming languages should allow the automatic checking of inconsistencies in programs. The most popular of these consistency checks is called static type checking (or static typing).

> We do static checking without any input to the program [12:13]

Static type systems help us detect programming errors even before we could run our programs. It is not just the simple mistakes (like forgetting to convert a string to a number) that can be caught by a type checker. A language with a rich set of types offers the opportunity to encode complex information about structure in terms of types but it requires attention and willingness from the programmer to make good use of the language's facilities.

> The strength of static typing is that it guarantees the absence of type errors in the programs it accepts. The weakness of static typing is that it rejects some programs that are in fact correct, but too complex to be recognized as such by the type system used. From this tension follows the search for more and more expressive type systems […] [30:3]

Static type systems are "conservative", meaning that they sometimes reject programs that actually behave well at run time. For example

```
if <complex test> then 5 else <type error>
```

will always be rejected as ill-typed even if `<complex test>` always evaluates to true, because static analysis cannot determine that this is the case. [38] For this reason, they are often criticized as too rigid: static type systems might reject programs that never produce type errors in practice.

> [static typing] consists in detecting a large family of errors: the application of operations to objects over which they are not defined […] This is achieved by grouping the objects handled by the program into classes: the types, and by abstractly simulating the execution at the level of types, following a set of rules called the type system. [30:3]

Static languages usually don't allow rebinding a variable to an object of a different type during run-time. This is what makes static type checking possible (and effective). [63]

3.2.1 Advantages of static languages

3.2.1.1 Peace of mind

Static type checking provides a basic set of guarantees about our code. We know that if it compiles, we used the types correctly. This is not a proof of correctness in general as well-typed programs can easily contain logic errors, bugs that cause the program to compute incorrect results even though it is a syntactically valid program. Logic errors are mistakes in the implementation that static typing cannot fix for us but they can provide us a safety net against a whole class of errors.

A static type checker checks the whole of the program in contrast with a dynamic type checker that only checks the parts that actually get executed when the program is running. Especially when changing parts of large programs, static checking makes sure we don't break something elsewhere in the code without having to rely on meticulous testing.

3.2.1.2 A more thoughtful design process

> Compiler-imposed constraints on data types encourage rigorous coding and precise thinking. [41]

When we encounter a static type error we stop and think about the inconsistency we were about to introduce. That record type is missing a field? Should I pass just an index instead of the whole list? This function is working with arrays but I was building a linked list in that other one? Type checker warnings provide useful guidance during development and greatly reduce debugging time.

> Good types lead to better code maintainability and faster failure on bad code [1]

3.2.1.3 Performance

> Static typing guarantees certain properties and invariants on the data manipulated by the program; the compiler can take advantage of these semantic guarantees to generate better code [31:1]

To generate efficient machine code, precise knowledge about the size of the data is required. This can be derived from static type information - memory size and layout. Languages without static typing cannot be compiled as efficiently, all data representations must fit a default size. [31]

> In general, accurate type information at compile time leads to the application of the appropriate operations at run time without the need of expensive tests. [8:6]

If a language's type system doesn't allow casts between incompatible pointer types, then those can't point to the same location in memory which in turn guarantees that load/store operations through those pointers cannot interfere. This allows the compiler to do more aggressive instruction scheduling. [31]

> Not having static type system information makes it hard to model control flow in the compiler, which leads to overly conservative optimizers. [15]

Type information is also useful in the optimization of method dispatch in object-oriented languages. [31]

> General method dispatch is an expensive operation, involving a run-time lookup of the code associated to the method in the object's method suite, followed by a costly indirect jump to that code. In a class-based language, if the actual class to which the object belongs is known at compile-time, a more efficient direct invocation of the method code can be generated instead. [31:1]

3.2.1.4 Documentation

> "The comment is lying!" - senior programmer

Source code comments usually get ignored when updating a code segment they refer to and since they don't have any meaning in the program, the runtime or the compiler can't give any warning about updating them: they "drift" from the code, often stating the exact opposite of what is implemented.

> Types are also useful when reading programs. The type declarations in procedure headers and module interfaces constitute a form of documentation, giving useful hints about behavior. [38:5]

Type annotations - explicit indications of types - are "living documentation". They provide information about the code and since they are verified by the type checker they cannot drift, they always stay up to date.

> Static type systems build ideas that help explain a system and what it does. […] They capture information about the inputs and outputs of various functions and modules. […] It's documentation that doesn't need to be maintained or even written. [43]

Note that static typing does not imply type annotations. Type-inferred languages (see "Type inference" later) do not need explicit indications of types in source code but they still handle types statically and so the documentation aspect of these languages is still valid because static type information can be used to generate enhanced documentation.

3.3 Dynamic type checking

> Even statically checked languages usually need to perform tests at run time to achieve safety. […] The fact that a language is statically checked does not necessarily mean that execution can proceed entirely blindly. [8:3]

In dynamically typed languages the types of program constructs don't have to be associated with a fixed type. Such languages let us, for example, define functions that can accept multiple types of values. The exact type of the arguments will only be known when the program is running and if the types don't match the operations

performed on them, they only report that at run-time. Such languages don't attempt static checks. In order to maintain type safety (prevent unintended program behavior) these languages instead perform various dynamic (run-time) checks to make sure that the data structures of the program stay consistent. Typical runtime checks include

- division-by-zero checks
- array bounds checking,
- verifying if a downcast is valid or not
- null pointer dereference check

When a dynamic check fails, the language runtime produces a runtime error. It depends on the language if a certain runtime error is recoverable or not: the language may allow the programmer to write error handling code and resume program execution after the error was handled. [53]

> We need to come to terms with persistency, inconsistency and change in programming languages. This means that dynamic programming languages should support the notion of software as living, changing systems, they should provide support multiple and possibly inconsistent viewpoints of these systems. Static type systems still have their place, but they should serve rather than hinder expressiveness. [36:10]

3.3.1 Advantages of dynamic languages

3.3.1.1 Quick and easy to write and read

Dynamic languages have rapid edit-run cycles (there is no upfront compilation before running the program). In the absence of a pre-runtime type checking phase, these languages favor prototyping and agile software development where creating proof-of-concept systems quickly are crucial. The code is free of type annotations and casting which makes it easier to write and read which can help maintainability.

3.3.1.2 Flexibility

A dynamic type system can help write more modular and decoupled code which may lead to a more flexible design. Dynamic typing encourages "generic code", code that is not "tightened down", free of fixed types that helps to think more abstractly. Module boundaries (the types, structures, interfaces that each module expects to see from another one) are not fixed down so it becomes possible to "plug in" other modules that may provide the required set of constructs without having to match them together. Modifying modules to meet interface requirements of other modules can be done incrementally.

> Dynamic languages are more tolerant to change; code refactors tend to be more localized (they have a smaller area of effect) [17]

4 Type systems in the real world

In this section I'll introduce type systems concepts from simpler to more advanced and how they appear in real, popular programming languages. There is a great deal of difference in how static and dynamic languages make use of types. Some concepts might make no sense in one context, while others might be used in both so let's clarify a few things first.

For static languages, the whole point of having type systems is to help us put constraints on parts of our code and have some automated tool (the type checker) verify that our constraints hold across our program. To make the most of the type checker, we try to define as much of our program as we can in terms of types. Different static languages provide different facilities for defining types and connections between them.

Dynamic languages must know the types of their values to do run-time checks in order to avoid illegal or undefined operations. In dynamic languages this is done by attaching a type label to every run-time value.

In the following section, I'll dive into how certain type systems concept appear in programming languages and how we can leverage these facilities to increase correctness

in our code.

4.1 No types - most assembly languages

An assembly language is one where there is a strong correspondence between the language's instructions and the instructions of the machine's instruction set architecture (ISA). That is, most language constructs have a 1-to-1 mapping to a CPU instruction. Since there are many types of different CPU ISAs, there are also many different assembly languages. There are some so called typed assembly languages (TAL) mostly within academic circles, but most assembly languages are said to be untyped.

Even though assembly supports certain basic data types, even arrays, structures and functions, it is mostly about defining the memory layout of the values. In assembly, we can't associate operations with types. In short, there is nothing to type check. This is the main point of assembly: manipulating data at the lowest level, giving access to individual bytes to the programmer.

Also, there are assembly languages that were designed to be compiler targets rather than used by programmers directly. For such languages there is little benefit for complex typing.

Assembly languages are - barring some academic derivatives - untyped. The values that we can manipulate in the language are just byte sequences. There is no way of knowing what a register or memory address holds just by looking at the assembly code. A given bit pattern may have multiple valid interpretations as different types. A certain system call might expect an 8 byte integer value for printing, but an 8byte floating-point value can also be interpreted as an integer. Assemblers don't try to check for matching types.

4.2 Concrete vs abstract types

Most programming languages differentiate between concrete and abstract types. A concrete type corresponds to the concept of a "data structure", a collection of data

values and the relationships between them. A concrete type in a programming language precisely defines the memory layout of the data that objects of that type hold.

Abstract types correspond to the concept of an "abstract data type" (ADTs) which is a mathematical model of a type. An ADT defines a type by its behavior from the point of view of its user. [56], [13]

> An ADT's user need not know how the object it represents is implemented […] In addition to the intellectual leverage for programmers who can take bigger strides in their thoughts, it provides flexibility in modifying the ADT implementation [13:34]

An abstract type defines behavior. It defines an "interface", a set of "calls" to which the objects of that type respond to. It describes how to interact with that object. Abstract types manifest themself in a number of different ways in programming languages:

- interfaces
- abstract classes
- pure virtual functions

Abstract types are related to but should not be confused with the general concept of "abstraction".

4.3 Structural vs Nominal typing

Structural typing is a way of relating types based solely on their members. Structural typing requires that a type supports a given set of operations. [45]

Duck typing is a loose form of structural typing: the implementation must be provided at run-time, not necessarily at compile time. There are no explicitly defined interfaces in the language which makes for terse and concise code as well as minimal coupling between different modules. Unfortunately, it also hides the dependencies between classes. [35] The phrase comes from the American poet, James Whitcomb Riley's duck test: "if it walks like a duck, and quacks like a duck, then it probably is a duck".

> In duck typing a statement calling a method on an object does not rely on the declared type of the object; only that the object, of whatever type, must supply an implementation of the method called, when called, at run-time. [53]

Structural typing is in contrast with nominal typing. In the case of a nominally-typed language, a subtype must explicitly declare itself to be related to the supertype. Nominal is more strict than structural. [68]

4.4 Gradual typing

Gradual typing originates in Siek & Taha's [25] 2006 paper. The idea is that a language should provide static and dynamic typing at the same time with the programmer controlling the degree of static checking by annotating function parameters with types [25]. Gradually typed languages usually (but not always) have an interesting property: an unsound type system, as is the case with TypeScript:

> The designers of TypeScript made a conscious decision not to insist on static soundness. […] it is possible for a program, even one with abundant type annotations, to pass the TypeScript type checker but to fail at run-time with a dynamic type error […] This decision stems from the widespread usage of TypeScript to ascribe types to existing JavaScript libraries and codebases, not just code written from scratch in TypeScript. It is crucial to the usability of the language that it allows for common patterns in popular APIs, even if that means embracing unsoundness in specific places [3]

As Bierman et al. [3] put it, it is a valid choice to go with a deliberately unsound type system in the case of gradually typed languages. This makes it possible to use it on existing large codebases that might contain possibilities for type errors at runtime (maybe only in unrealistic scenarios) but that still benefit from the added static type checking. [67]

4.4.1 Optional type annotations in Python 3

Even though Python 3 is dynamically typed, the language allows optional "type hints" which are similar to type declarations in languages like Java. These annotations may be used together with type checkers like `mypy`, `pyre-check` or `pytype` [46]. The programmer is free to annotate only parts of the source code. The Python 3 runtime itself doesn't type check the type hints. Due to python's dynamic nature, the following code will be executed without any type errors in Python 3.5.2:

```
def f(a: int, b: int) -> str:
    return a + b

print(f("x", "y"))
print(f(1, 2))
```

The programmer has to use one of the external type checkers to make use of type hints. Running `mypy` on the above code indeed reports type errors complaining about an incompatible return type declaration and incompatible argument types when calling the function with integer values.

In my own experience, on one hand, gradual typing tends to make the programmer lazy. Omitting type annotations can help finish writing certain parts of the program faster. The program might even run without error when tested. The problem is that we likely didn't cover all the possible execution paths and by "shutting down" the type checker we made it harder to find the possible inconsistencies in the code. The program seems to behave correctly but had we defined types properly, we could have been warned about type errors that will now show themselves later at an unexpected time.

On the other hand, having the possibility of retrofitting a codebase with static type checking is always good. Adding some type safety a step at a time to an existing large code base by providing a smooth transition and thus not blocking development can provide great value.

4.5 Type inference

Type reconstruction or informally, type inference is the process of automatically (instead of manually, by the programmer) assigning types to expressions in a program by examining the operations that are performed on them.

> The first statically typed languages were explicitly typed by necessity. However, type inference algorithms - techniques for looking at source code with no type declarations at all, and deciding what the types of its variables are - have existed for many years now [43]

Type inference should not to be confused with dynamic typing. Even though types may not be visible in the source code in an inferred language, it might check those types statically (before execution).

The challenge of type inference is not assigning a type to untyped terms in a program, but rather to find the balance between the most general and most specific type that could have been declared in the program. [16]

By utilizing type inference, code written in the language can be more concise and more easily understandable. Having to spell out complex types can be a burden and can hurt readability and often provide no benefit. Type inference seems to be the golden mean: making static type checking possible while not enforcing the programmer to be explicit about their types.

4.6 Polymorphic typing - "Polymorphism"

A language, where every expression has a single type is called monomorphic. In such a language there is a distinct identity function of each type: $\lambda\ (x : \tau)\ x$ even though the behavior is exactly the same for each choice if τ. Although they all have the same behavior when executed, each choice requires a different program. [20]

> […] motivated by a simple practical problem (how to avoid writing redundant code), the concept of polymorphism is central to an impressive variety of seemingly disparate concepts […] [20:141]

Polymorphism (poly = many, morph = form) is the concept of having internally different objects that provide the same interface. Even though they implement different behaviour and have different structure, in some aspect they "look the same" from the outside and so they can be used interchangeably. It is the programmer's task to declare such similarities between different types. Of course it is not only about declaring similarity; the programmer has to create some structure like implementing a common method or placing the types in the same hierarchy or wrapping them with the same structure. The way we do this is by using some sort of polymorphism that the language at hand provides. The point of the whole thing is to decaler similarity and interchangeability on the type level so that we can utilize the type checker.

To facilitate code-reuse, most programming languages feature a polymorphic type system. A polymorphic languages are those of which the type systems allows different data types to be handled in a uniform interface. [47] Stated more simply: in a polymorphic language, a program fragment may have multiple types. There are different kinds of polymorphisms, I will look at each of them in more detail below.

4.6.1 Ad-hoc polymorphism - overloading

Ad-hoc polymorphism means polymorphic functions, that is, functions that can be applied to different types of arguments and the implementation depends on the types of the arguments. Ad-hoc here means that this is not a feature of the type system but instead a mechanism that is about using the same name for different (but related) operations. [57] The typical examples are method or operator overloading in most OOP languages. Method overloading is the provision of different implementations depending on the type of arguments the method (function) is called on and this can happen statically or dynamically. [38]

4.6.2 Subtype polymorphism - Subtyping

Subtype polymorphism or runtime polymorphism is the process of taking a type (the base type) and defining a more specialised type (the subtype) based on it by extending

it with functionality or overriding some parts to facilitate code reuse. The subtype is related to the supertype by the notion of substitutability which means that in a program the subtype should be able to be used wherever the supertype is expected. [58] In most classical object oriented languages subtyping is tied to inheritance in the form of class hierarchies. Class hierarchies together with method overriding (providing different or extended implementations in child classes) is the primary vehicle of subtype polymorphism that is made possible by dynamic method dispatch.

Interfaces in OOP languages also implement something like subtype polymorphism: they provide a common type for types that may not otherwise be related.

4.6.2.1 Difference between subtyping and inheritance

Go is a good example of how subtyping is a different concept from inheritance. In Go, structs (a collection of related attributes) with methods resemble classes of classical OOP languages but in Go there isn't any kind of polymorphism for structs. Struct types can embed other struct types where the embedder type gains access to the methods of the embedded type. This provides a limited form of inheritance through composition. The embedder type however is not polymorphic with the embedded one (it doesn't become a subtype of the embedded one).

In Go, we use interfaces to create subtype relationships. An interface is a set of function signatures and any type that implements all the functions ("methods") that an interface declares is said to implicitly implement that interface - also a case of structural (instead of nominal, as in most OOP languages) subtyping. A type that implements an interface is considered a subtype of that interface (similarly to most OOP languages).

```
type Position struct {
    X, Y int
}

type Animal struct {
    Position
    isAlive bool
```

```
}

type Bunny struct {
    Animal
}

type Entity interface {
    Pos() Position
    Move()
    Die()
}

func (a *Animal) Pos() Position {
    return a.Position
}

func (a *Animal) Move() {
    // ...
}

func (a *Animal) Die() {
    a.isAlive = false
}

func needAnimal (a Animal) {}

func needEntity (e Entity) {}
```

The above is a short Go code snippet to demonstrate interface-based polymorphism in the language. It defines 3 struct types: `Position` , `Animal` and `Bunny` and an interface type `Entity`. `Bunny` embeds an `Animal` and thus gains access to its methods and fields so code reuse is achieved (similar to inheritance). `Bunny` doesn't become a subtype of `Animal` however which means a `Bunny` may not be given where an

`Animal` is expected. `Animal` on the other hand does become a subtype of `Entity` by implementing all its methods and so it can be used wherever an `Entity` is expected. `Bunny` also becomes a subtype of `Entity` because it embeds an `Animal` which itself is an `Entity`. Also notice type inference in action, as I didn't say what's the type of `someAnimal` or `bob` and yet the type checker correctly infers them:

```
func main() {
    someAnimal := Animal{Position{1,2}, true}
    bob        := Bunny{someAnimal}

    // bob is not an Animal -> type error
    //needAnimal(bob)

    // however, bob can move
    bob.Move()

    // someAnimal is an Entity
    needEntity(&someAnimal)

    // bob embeds an Animal which implements Entity
    // so bob is an Entity
    needEntity(&bob)
}
```

4.6.3 Parametric polymorphism

Sometimes called "compile-time polymorphism", parametric polymorphism means that concrete types are abstracted away from the implementation and type parameters (or type arguments) are used instead. This way static type checking can still be done based on type variables while providing flexibility since they can take on any type as long as that type is used consistently within the scope of the type parameters. The simplest use case for parametric polymorphism is implementing generic "container" types where the emphasis is not on the type of the values in the container but instead on

their consistent use within the container (generic lists or tuples). The type checker can aid in enforcing the consistent usage without dictating the exact types of the values. Type parameters help us encode the notion of composition on the type level.

The Java language (along with C#, Visual Basic and Delphi) calls their implementation of parametric polymorphism "Generics" and it looks something like this:

```
LinkedList<String> list = new LinkedList<String>();
list.add("abc");       // fine
list.add(new Date());  // error
```

Here, the `LinkedList<T>` class is a "generic class" because its definition includes a type parameter. Each time the class is used, a concrete type has to be used in place of the type parameter. From that point on, that instance of the class is bound to that type and there is no need to cast the objects in the list even though their type was not part of the class definition.

Without a generic class:

```
LinkedList list = new LinkedList();
list.add("abc");       // fine
list.add(new Date());  // fine as well
```

This `LinkedList class` is not generic. In Java, all classes inherit from the `Object` class so this non-generic linked list treats every one of its items as if it was an `Object` instance which means, statically it doesn't know anything about their specific types, so only methods and properties of the `Object` class are accessible on them unless we explicitly cast the item to a specific type. [29]

In Elm, the same concepts apply but since there are no classes, parametric polymorphism can be found on the function level where function arguments have type parameters. The following is the type signature of the `List.map` function that implements the usual "map" operation (applying the same transformation for each element in a list) on lists in a generic way:

```
map : (a -> b) -> List a -> List b
```

The lowercase names (only single letters in this case) are type parameters. This signature says that the `map` function takes 2 parameters: the transformation function that takes an `a` and returns a `b` and a `List` of `a`s on which to apply the transformation. The return type is a `List` of `b`s. I find this syntax very elegant. Thanks to the type parameters, the type checker can verify that the transformer function's input matches the input list's type and its output matches the output list's type without dictating any concrete types.

We could map a list of strings to their lengths using the generic `List.map` function above:

```
words = ["type", "systems", "are", "fun"]
wordLengts = List.map String.length words
```

4.7 Reflection

> Many programming languages require compiled programs to manipulate some amount of type information at run-time [31:4]

Reflection is an umbrella term for programming language features that allow us to inspect and modify a program while it is running. It is an inherently dynamic process and so it conflicts with many of the principles of statically typed languages. Most static programming languages do provide reflection facilities, but with limited support because it is essentially a way of circumventing the static type system.

> Reflection enables the changing of systems without the need to rebuild or even restart them. This is an important basis for building the dynamic systems of the future: Mobile, Ubiquitous, Always-On. [36:6]

When used cautiously, reflection can give the programmer power over parts of a program that would not otherwise be accessible by the runtime like internals of private/protected components or 3rd party libraries. However, a totally reflective (dynamic) system with "unscoped" reflection suffers from many disadvantages:

- Security: The clients that use reflection can do anything

- Stability: The effects of reflection are global. One client using reflection affects the other clients
- Performance: Full reflection is costly.

To solve the above issues, the reflective capabilities of the language must be scoped:

- define when and where reflection should be available
- limit the reflective interface to certain clients
- constrain the effects of reflection

[36]

Reflection can help with serialization (or "marshalling") of objects when the objects themselves were originally not designed to support it and it would be too tedious to prepare a hierarchy of classes for serialization. Debuggers or class browsers can be created using reflection. Object-relational mappers (ORM) that create object oriented abstractions of database entities are popular in business software and are another use case for reflection. Testing frameworks that are based on naming conventions also depend on reflection to dynamically discover methods which follow a certain naming pattern.

4.8 Variance

The rules that govern how subtyping between complex types relate to subtyping between their component types are called variance. Within the type system of a programming language, a typing rule can be

- covariant if it preserves the ordering of types ($\leq$), which orders types from more specific to more generic: if `A` $\leq$ `B`, then `I<A>` $\leq$ `I<B>`;
- contravariant if it reverses this ordering: if `A` $\leq$ `B`, then `I<B>` $\leq$ `I<A>`;
- bivariant if both of these apply

[59]

4.8.1 Function parameter bivariance in TypeScript

TypeScript is a gradually typed superset of the JavaScript language with optional type annotations and provides a type checker and transpiler for JavaScript programs. Thanks to its type inference, every valid JavaScript program is also a valid TypeScript program so it is possible to gradually transform a JavaScript codebase into a TypeScript one. TypeScript's static type system is unsound by design to allow for "backwards compatibility" and an incremental transition from dynamically typed JavaScript codebases.

One example of unsoundness in TypeScript is "function parameter bivariance":

> When comparing the types of function parameters, assignment succeeds if either the source parameter is assignable to the target parameter, or vice versa. This is unsound because a caller might end up being given a function that takes a more specialized type, but invokes the function with a less specialized type. [68]

```
enum EventType {
  Mouse,
  Keyboard,
}
interface Event {
  timestamp: number;
}
interface MyMouseEvent extends Event {
  x: number;
  y: number;
}
interface MyKeyEvent extends Event {
  keyCode: number;
}
function listenEvent(
    eventType: EventType,
```

```
    handler: (n: Event) => void
) {
  /* ... */
}
// Unsound, but useful and common
listenEvent(
    EventType.Mouse,
    (e: MyMouseEvent) => console.log(e.x + "," + e.y)
);
```

The above snippet is from TypeScript's official documentation ([68]) and it demonstrates how an unsound type system might even be desirable in their case to be able to work with existing JavaScript code. Here, the `MyMouseEvent` interface is a subtype of the `Event` interface. The `listenEvent` function declares its second argument to be a function that takes an `Event` and returns nothing. Yet, when called, it accepts a function that takes a `MyMouseEvent`, a more specialized type (the compiler flag `strictFunctionTypes` must be turned off for this to work). The `listenEvent` function can call its `handler` function with an object of the base type `Event`, because that is how it was defined. An `Event` object will not have an `x` and `y` field so the supplied callback in the example will result in a runtime error in those cases. TypeScript allows code to be written that can not be proven to work correctly under all circumstances. Such circumstances might be rare in practice and it allows common Javascript patterns like the above example.

4.9 Algebraic data types

Algebraic data types are composite types: they are defined as a combination of other types. There are two main classes of algebraic data types: product types and sum types. Product types are structures that can hold more than one value in a single structure at the same time. This is just a fancy name for common and simple programming language constructs like tuples or records (maps or structs). Product type are not particularly interesting from a type systems standpoint but sum types very much are. [60] [42]

4.9.1 Sum types

To understand sum types (also called tagged unions or variant types), first let's define union types. Union types are usually denoted as `A | B` where `A` and `B` are some types. A union type between two types `A` and `B` is simply the union of the two sets of values, so it may hold any value from either `A` or `B`. For example, the union type `null | int` can hold `null` or any integer value. [26]

Sum types are very similar to union types in that they hold a value that must be one of a fixed set of options.

> Only one of the types can be in use at any one time, and a tag field explicitly indicates which one is in use. It can be thought of as a type that has several "cases," each of which should be handled correctly when that type is manipulated. [61]

Every type in the sum type is accompanied by a label (or tag), hence the name "tagged union". This label is a unique identifier for this element of the sum type. A sum type can also be thought of as an enum, with a payload where that payload is the actual value that it holds. [26] [50]

> [A tagged union is] a union, but each element remembers what set it came from [26]

As Chad Austin [6] puts it: a sum type is a combination of a tag (like an enum) and a payload per possibility (like a union). Sum types are a safe generalization of the two:

```
enum EventType {
  CLICK,
  PAINT
};

struct ClickEvent {
  int x, y;
};
struct PaintEvent {
```

```
  Color color;
};

struct Event {
  enum EventType type;
  union {
    struct ClickEvent click;
    struct PaintEvent paint;
  };
};
```

The above is an example of a sum type implementation in C by Chad Austin[6]. The `EventType` enum serves as the tags of the sum type and the union of the `ClickEvent` and `PaintEvent` serve as the payload. A very important weakness of this C code is that nothing prevents the programmer from accessing the `.paint` field on a `CLICK` event or conversely the `.click` field on a `PAINT` event and both would lead to subtle bugs that the compiler couldn't warn us about. This is an unsafe implementation of sum types.

Here is the definition of the same sum type in Elm (an language of the ML family):

```
type Event
  = ClickEvent Int Int
  | PaintEvent Color
```

I especially like the ML syntax because it is intuitive: this must be some event that can take on certain forms where each form contains different data about the event.

What if the language forced the programmer (with convenient syntax and the use of its type system) to write code for each possible variant of the sum type and only allow access to its payload where it makes sense? This is what languages in the ML family do with their `case` expressions, a form of so called "pattern matching":

```
case event of
  ClickEvent x y ->
    handleClickEvent x y
```

```
  PaintEvent color
    handlePaintEvent color
```

The above Elm snippet is a `case` expression. The language enforces that these expressions go through all variants of the used type so the programmer can not forget to check some condition. It also makes sure that the `x` and `y` coordinates of a `ClickEvent` are only accessible within the scope of the `ClickEvent`. This makes it a robust base for an error handling model.

The following Elm code shows how the programmer is forced to handle all the possible types of HTTP errors:

```
httpErrorToString : Http.Error -> String -> String
httpErrorToString err errorMessagePrefix =
  case err of
    Http.BadUrl message ->
      message
    Http.Timeout ->
      "Network timeout"
    Http.NetworkError ->
      "Network error"
    Http.BadStatus statusCode ->
      String.fromInt statusCode
    Http.BadBody response ->
      response
```

Another great use case for sum types is “null-tracking”:

> The compiler statically ensures that null values are handled explicitly, instead of allowing failure at runtime with the equivalent of a null-pointer exception [26]

The need for null-tracking stems from the ever-present `NullPointerException` errors which in turn are the result of Tony Hoare’s famous “billion dollar mistake” [22] of allowing types to have a “null” value.

There is nothing wrong with the concept of "nothing" or "no value", that is a very useful thing in programming. The problem of `null` in most imperative languages is that values may be `null` without it being obvious from the code and so programmers can easily forget to check those `if (x == null) { ... }` cases which lead to run-time errors.

Sum types are the basis of what some languages call the "Option type" or "Maybe type":

```
type Maybe a
  = Just a
  | Nothing
```

The above is the built-in `Maybe` type of Elm. Notice that it combines parametric polymorphism with sum types, the lowercase `a` there represents a type parameter which may be any concrete type. It is a type that says "I either have some value of type `a` or I have `Nothing`". The key to this type (and the language) is that the programmer can not directly access the underlying value. Pattern matching must be performed and that forces to programmer to also handle the case where it holds a `Nothing` (the equivalent of a null value in other languages). Unlike `null` in most imperative languages, `Nothing` is not compatible with other types so it can not be returned from functions. Functions where the result may be a "no value" must use a `Maybe` type as their return type and so the callers of these functions are forced to explicitly handle the "no value" cases.

In theory, a language that supports `Maybe` or `Option` types (parametric sum types) and enforces explicit handling of the `Nothing` case is a language where a `NullPointerException` is no longer possible. Tony Hoare called the null value the "billion dollar mistake" because he believes that the amount of man-hours wasted examining and fixing software bugs related to invalid pointers could already be translated to losses of that magnitude for the software industry. If we believe Prof. Hoare's estimate to be correct, it would make a lot of sense to start using languages and libraries that are built around the notion of sum types.

> When nullable references are replaced by explicit Maybe or Option, you

> no longer have to worry about NullPointerExceptions, NullReference-Exceptions, and the like. The type system enforces that required values exist and that optional values are safely pattern-matched before they can be dereferenced. [6]

The utility of sum types doesn't end here. These wonderful constructs are a great tool for implementing error handling which is concerned with the problem of how to signal to the caller of your function that something went wrong. [10]

Various error handling schemes exist in programming languages, the most basic probably being "sentinel values". In the case of "sentinel values", some values in the range of the function (as defined by its return type) are interpreted as having a special meaning. This has the advantage of being very simple to implement (from the language's point of view). It is also very easy to forget to handle these values - the errors can go unnoticed.

> Values that are considered impossible as a calculation result are repurposed as error indicators. […] Due to a lack of type safety or by faulty or no type checking, we can accidentally use return values indicating errors as if they were computation results.
>
> [21]

Then there are exceptions, which are a mechanism based on automatically propagating some event up the call chain until it is handled explicitly or until it reaches the top at which point the runtime handles it by issuing some kind of runtime error.

The exception mechanism has its weaknesses too. It makes error handling more implicit (execution may continue at unexpected places), it can be overused to implement branching logic and handling them is usually not enforced (checked and unchecked exceptions in Java).

Some languages like Go, allow multiple values to be returned from functions where one value indicates the error (if any) and the other value holds the result of the computation (if all went fine). However, without sum types and explicit pattern matching enforced by the type checker, these error values are easy to ignore and forget about

handling the potential errors. Also, when a Go function returns for example a (string, error) pair to signal that it may fail, nothing stops the programmer from setting both values which is a logic error but can not be discovered by the compiler.

A more elegant and bulletproof way of error handling is to use sum types as return values (sometimes called "Monadic error handling"):

```
enum Result<T, E> {
   Ok(T),
   Err(E),
}
```

The above is the definition of Rust's `Result` type which is also a parametric sum type like `Maybe` in Elm but here two type parameters are used instead of one. The `OK(T)` variant is for the case where the given computation succeeds and the result is valid. The `Err(E)` variant is for the case where an error needs to be reported. [5] Since the `Err(E)` variant of a `Result` may not be ignored, it is especially useful with functions that may encounter errors but don't otherwise return a useful value. Such is the `write_all` method defined for I/O types in Rust:

```
fn write_all(&mut self, bytes: &[u8]) -> Result<(), io::Error>;
```

Notice how the first type parameter is the "unit type" `()` which doesn't hold any value and so no value will be extracted when pattern-matched but the second type parameter is an io::Error so that the cause of the error can still be extracted. [69]

In most imperative languages conditional expressions (if-else statements) can define any number of branches that are not checked for consistency. Execution enters these branches based solely on their predicates, boolean valued "functions". This means that by mistake, they can overlap or fail to handle all possible cases. In functional languages, with the help of pattern matching, sum types facilitate a type safe implementation of conditional expressions. They let us define the branching logic in terms of a composite type and allow the type system to check whether we covered all the cases. [37]

4.10 Ownership

The ownership system is one of the main innovations of the Rust programming language, whereby it can statically determine when a memory object is no longer in use. Through ownership, Rust provides memory safety guarantees at compilation time.

> All programs have to manage the way they use a computer's memory while running. Some languages have garbage collection that constantly looks for no longer used memory as the program runs; in other languages, the programmer must explicitly allocate and free the memory. Rust uses a third approach: memory is managed through a system of ownership with a set of rules that the compiler checks at compile time. [64]

Memory safety means that the software never accesses invalid memory. Such invalid memory accesses could be use-after-free, null pointer dereferencing, using uninitialized memory, double free or buffer overflows [23], all of which are common causes of severe bugs and vulnerabilities. Automatic memory management is available for most high level languages in the form of garbage collection (GC), but it might not be a feasible solution under every circumstance: programs written for embedded systems with very constrained resources can't afford to keep allocating memory and wait for a garbage collector to kick in. Similarly, systems with real-time requirements can't be stopped while the garbage collector does its work. In these scenarios the programmer is often left with the only choice of doing manual memory management and hope for the best.

The problem with manual memory management is that it puts the burden on the programmer and deallocating memory at the right time is a difficult problem. Too conservative memory management might result in use-after-free types of errors while being too generous with memory is the source of memory leaks. Garbage collection solves the problem of safety but it makes the program unpredictable which might not be affordable in scenarios like real-time systems. [34] Providing memory safety without sacrificing performance is considered to be one of the next big challenges of programming language design and rust might have an answer to this problem with its ownership system.

Rust's ownership system is based on a technique called region inference. Region inference is a memory management discipline. It is a technique for determining when objects become dead by a static analysis of the program [70]. The ML Kit compiler (a compiler from Standard ML to assembly) features a so called "region inference algorithm" which is a static analysis technique to examine the lifetime of dynamically allocated values in a program that makes it possible to replace garbage collection with stack-based memory management. [34]

In region inference's runtime model the store (memory) consists of a stack of regions. At runtime, all values are put into regions. All decisions about where to allocate and deallocate regions are made statically, by region inference. For every value-producing expression, region inference decides into which region the value is to be put. [49] [34]

> In some cases, the compiler can prove the absence of memory leaks or warn about the possibility of memory leaks [34:725]

Aliasing is the situation when a location in memory can be accessed through different symbolic names in a program. Modifying the data through one name implicitly modifies the values associated with all aliased names which may not be expected by the programmer. Such implicit changes during the execution of a program make it very difficult to understand and reason about. [11] [62] Controlling and statically checking aliasing is the main theme of the ownership system and the so-called "borrow checker" in Rust. The borrow checker is the component in the Rust compiler that is responsible for enforcing the rules of the ownership system. The ownership system is based on the following 3 rules [71]:

- each value in Rust has a variable that's called its owner
- there can only be one owner at a time
- when the owner goes out of scope, the value will be dropped

In contrast to C for example, there are no explicit `free()`-s and `malloc()`-s in Rust code. Memory is automatically returned once its owner goes out of scope. The language handles variable assignments in different ways depending on the types of the values being assigned. If the size of the value is known statically at compile time,

then it is stored on the stack of the process's virtual memory but if it is not known at compile time, it is stored on the heap. To demonstrate the rules of ownership, let's take a look at two simple, very similar examples. Values of `String` type in Rust are mutable and have dynamic lengths (in contrast with string literals) so they are stored on the heap. They are a simple example of structures that have an unknown size at compile time. Such values are relatively expensive to copy because of heap management operations so when a String value is assigned to a variable, it is said to be "moved" instead of copied:

```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```

Rust calls it a move because the variable that was previously the owner of this `String` value is invalidated as soon as the assignment `let s2 = s1;` happens and from that point on accessing `s1` is a compile time error:

```
let s1 = String::from("hello");
    -- move occurs because `s1` has type `String`,
    which does not implement the `Copy` trait
let s2 = s1;
         -- value moved here

println!("{}, world!", s1);
                       ^^ value borrowed here after move
```

> When a variable that includes data on the heap goes out of scope, the value will be cleaned up by `drop` unless the data has been moved to be owned by another variable [72]

On the other hand, values stored on the stack are relatively cheap to copy so Rust creates copies of such values when they are assigned to a new variable:

```
let x = 5;
let y = x;
```

```
println!("x = {}, y = {}", x, y);
```

The above code is valid, because `x` and `y` are two distinct owners with two distinct values. The same rules apply for passing a value to a function. It will either move or copy the value just as assignment does. In case of a move, the ownership of the value is transferred to the function's local variable that was declared as its input parameter. This means that if we don't also return it from the function (even if it doesn't modify it) then we lose the value in the outer scope where we called the function. This is where references and the concept of "borrowing" come in.

Borrowing is the action of creating a reference. References allow us to refer to a value without taking ownership of it:

```
fn calculate_length(s: &String) -> usize {
    s.len()
}
```

Here, the argument is of type `&String` which is a reference to a String. `s` inside the function doesn't have ownership of what it refers to so even though it goes out of scope when the function returns, the data it references is not dropped. References are immutable by default meaning the `calculate_length` function is not allowed to modify the value `s` refers to. To make a reference mutable, we prefix it with the `mut` keyword but there can only ever be a single mutable reference to a particular piece of data at a time. We can have any number of immutable references to data but as soon as we have mutable reference, having immutable references is not allowed. [72]

These restrictions make it possible for Rust to eliminate data races which are similar situations to race conditions:

- two or more pointers access the same data at the same time
- at least one of the pointers is being used to write to the data
- there's no mechanism being used to synchronize access to the data

  Data races cause undefined behavior and can be difficult to diagnose and fix when you're trying to track them down at runtime; Rust prevents this

> problem from happening because it won't even compile code with data races! [72]

The concept of a dangling pointer must be very familiar to C/C++ developers. A pointer becomes dangling when the object it points to is deallocated but the pointer itself is not updated. This situation is impossible in Rust thanks to its compile-time borrowing rules.

Rust's standard library (a large set of functions that provide interfaces to the OS kernel) is optional, which means it is suitable for writing programs that will run on platforms without operating systems: embedded systems with minimal resources. Just like C or C++, Rust also gives the programmer fine-grained control on when and how memory is allocated which helps create programs with predictable performances. [32]

> 70% of security vulnerabilities that Microsoft fixes and assigns a CVE (Common Vulnerabilities and Exposures) are due to memory safety issues. This is despite mitigations including intense code review, training, static analysis, and more. [33]

Development teams at Microsoft started adopting Rust for systems programming tasks as an alternative to C/C++ to mitigate the huge cost that memory-safety bugs can bring [33]. Dropbox rewrote their sync engine in Rust [24]. Cloudflare wrote BoringTun, an open source WireGuard VPN implementation in Rust so that it is based on a solid memory model that matches the needs of a modern cryptography and security-oriented project [27]. The Oxide Computer Company on their mission to reinvent the physical server is also betting big on Rust: their software systems (firmware, embedded systems) are almost exclusively written in Rust and they are very pleased with their results [7].

5 Suggestions for further research

The following topics are closely related to the theme of this work and could be interesting continuations.

5.1 Dependent types

Dependent types are based on the idea of using scalars or values to more precisely describe the type of some other value. [53] Dependent types can express for example that an `append` function that takes two lists of length `m` and `n` should return a list of length `m+n`. Similarly, the rules of matrix multiplication can be expressed:

$$\frac{\Gamma \vdash A : matrix(l, m),\ \Gamma \vdash B : matrix(m, n)}{\Gamma \vdash A \times B : matrix(l, n)}$$

Which we read as "if A is an $l \times m$ matrix and B is a $m \times n$ matrix, then their product is an $l \times n$ matrix".

With dependent types, a large set of logic errors can be ruled out statically. Dependently typed languages are rare and are mostly academic in nature. The most well known languages that feature dependent typing are Coq, Agda and Idris.

5.2 Linear types

Linear types are the manifestation of Jean-Yves Girard's "linear logic" in programming languages. It is about modeling resource usage and inferring statically how many times resources are used; "linearity" in this case means "used exactly once" which is a useful concept when talking about memory allocation and deallocation. Walker and Watkins' paper [51] explores the connection between regions, effects and linear types. There is ongoing work to extend Haskell with linear types. [73]

5.3 Effect systems

Effect systems aim to model the side effects of computation. Resorting to the definitions of functional programming languages, a "pure function" is a unit of code whose output depends only on its inputs and doesn't cause any observable effect besides returning a value. In contrast, an "effectful" or "impure" function is one that can cause

some observable effect besides its return value: perform I/O or - and this might sound odd - fail. Failure here means that some runtime mechanism is triggered to handle some unexpected condition, this is usually implemented as exceptions. A function that doesn't terminate (infinite loop or recursion) is also considered effectful. Determining such properties of functions and checking their consistent use could help produce more bug-free software and this is the goal of effect systems. Such features are not yet found in mainstream languages but Microsoft Research's Koka language is an example of a research project in this direction. [14] [74]

6 Summary

In the first half of my thesis, I looked at type theory to lay down the fundamentals of type systems. Then I talked about what it means for a language to be type safe or "sound" and then examined the two main types of type checking languages perform to achieve it: static and dynamic checking. In the second half, I examined a list of type systems concepts from simpler to more advanced, illustrated them with examples of common programming languages and talked about their importance and potential to increase the quality of software produced by the industry.

From the various type systems features I discussed above, the ones that I find the most important are:

- gradual typing
- sum types
- Rust's ownership system

Gradual typing for its potential to help existing large codebases slowly transition to static type checking, sum types for their ability to elegantly express and model uncertainty, their excellent fit for implementing robust error handling and their role in eliminating runtime errors and finally Rust's ownership system for its potential to bring static memory safety closer to the mainstream and provide an alternative to low-level but unsafe systems programming languages.

Part of my goal was to gather evidence that shows that static typing is superior to dynamic typing for large scale software development. I am still convinced that a statically typed language is a better tool for writing correct, high quality software but I am surprised by the lack of evidence that can support this claim. In fact, all the studies I could find conclude that there is no objective, measurable difference between the quality of software produced with dynamically or statically typed languages [40]. This suggests that it is a matter of personal taste if someone feels more productive or confident with one or the other. However, many dynamic languages seem to be adding support for some form of gradual typing, this suggests a dissatisfaction with dynamic typing.

In my view - which might change in the future - dynamic languages do have their place which is small-scale sofware development, typically small utilities, automation scripts or proof-of-concept software that are not expected to grow much (if only we could know…). The small size usually means a smaller “input space”, less execution paths and less things in which a static type system would show its real value and on the other hand, being free from type constraints means we can get usable results more quickly. Dynamic languages also offer a low barrier to entry for people new to programming which is also a crucial role to fill.

7 Összefoglaló

Szakdolgozatom első felében áttekintettem a típuselméletet, hogy lefektessem a típusrendszerek alapjait. Ezt követően bemutattam, mit jelent a típusbiztosság illetve a típusellenőrzés két fő fajtáját, a statikus és dinamikus típusellenőrzést. A dolgozat második felében megvizsgáltam számos típusrendszer fogalmat az egyszerűbbtől a kifinomultabbig, ismert nyelvek kódrészletein keresztül bemutattam ezeket és kitértem a fontosságukra és szerepükre a szoftverminőség tekintetében.

A különböző típusrendszer koncepciók közül a következőket tartom legfontosabbak:

- graduális típusosság
- algebrai típusok
- a Rust nyelv ownership rendszere

A graduális típusosságot, mert kiváló eszköze lehet a nagy, dinamikus nyelven íródott kódbázisok fokozatos statikus ellenőrzésre való átvezetésének, az algebrai típusokat, mert elegánsan kifejezhető és modellezhető velük a bizonytalanság, jól alkalmazhatóak robusztus hibakezelés megvalósítására illetve komoly szerepet töltenek be a futásidejű hibák eliminálásában és végezetül a Rust ownership rendszerét a statikus memóriakezelésében rejlő potenciálért és mert remek alternatívát kínál az alacsony szintű de nem biztonságos rendszerprogramozó nyelvekre.

Dolgozatom egyik célja az volt, hogy alátámasszam, a statikus nyelvek alkalmasabbak nagy szoftverek fejlesztésére, mint a dinamikus nyelvek. Továbbra is meggyőződésem, hogy a statikus nyelvek jobb eszközök helyes, magas minőségű szoftverek készítésére, ám meglep, hogy ezt kutatások nem támasztják alá. Az általam vizsgált kutatások közül egy sem talált objektív, mérhető különbséget a dinamikus és a statikus nyelveken írt szoftverek között a minőség vagy a fejlesztés sebessége tekintetében [40]. Ez arra enged következtetni, hogy pusztán személyes preferencia kérdése, hogy ki mely irányzatot követve érzi produktívabbnak magát. Ugyanakkor számos dinamikus nyelvhez készülnek graduális típusokat felvonultató variánsok és statikus típusellenőrzők, ezek elszaporodása a dinamikus nyelvek hiányosságaira utalnak.

Saját véleményem - ami könnyen változhat a jövőben -, hogy a dinamikus nyelveknek is megvan a helyük, leginkább a kis támogató szoftverek, automatizáló scriptek vagy kísérleti jellegű szoftverek fejlesztésében, amelyek várhatóan nem nőnek nagyobbra egy bizonyos szintnél (bár tudnánk ezt előre…). A kis méret általában kis "input teret" is jelent, ez kevesebb lehetséges végrehajtási utat, vagyis kisebb a statikus típusellenőrzés hozzáadott értéke, viszont a laza típuskezelés által hamarabb juthatunk használható eredményhez. Ezen kívül a dinamikus nyelvek egy alacsony belépési korlátot adnak azoknak, akik még csak ismerkednek a programozással, ez pedig egy rendkívül fontos szerep.

References

[1] Joseph Abrahamson. 2014. What are the programming languages with the best type system? Why? Retrieved July 30, 2019 from https://www.quora.com/What-are-the-programming-languages-with-the-best-type-system-Why/answer/Joseph-Abrahamson?ch=10&share=6b0f63a3&srid=8nW0V

[2] Cardelli at al. 1995. An imperative object calculus. *CAAP 1995: TAPSOFT '95: Theory and Practice of Software Development* (1995).

[3] Bierman et al. 2014. Understanding typescript.

[4] Monnier et al. 1998. Implementing typed intermediate languages. *ACM SIGPLAN Notices, September 1998* (1998).

[5] Tony Arcieri. A quick tour of rust's type system: Part i: Sum types aka. tagged unions. Retrieved November 1, 2021 from https://tonyarcieri.com/a-quick-tour-of-rusts-type-system-part-1-sum-types-a-k-a-tagged-unions

[6] Chad Austin. 2015. Sum types are coming: What you should know. Retrieved October 28, 2021 from https://chadaustin.me/2015/07/sum-types/

[7] Bryan Cantrill. Rust after the honeymoon. Retrieved December 1, 2021 from

http://dtrace.org/blogs/bmc/2020/10/11/rust-after-the-honeymoon/

[8] Luca Cardelli. 1996. Type systems. In *Handbook of computer science and engineering*, Allen B. Tucker (ed.). CRC Press.

[9] Glenn G. Chappell. 2015. A primer on type systems. Retrieved May 2, 2019 from https://www.cs.uaf.edu/users/chappell/public_html/class/2018_spr/cs331/docs/types_primer.html

[10] Dave Cheney. 2012. Why go gets exceptions right. Retrieved from https://dave.cheney.net/2012/01/18/why-go-gets-exceptions-right

[11] David G. Clarke, John M. Potter, and James Noble. 1998. Ownership types for flexible alias protection. *SIGPLAN Not.* (1998). Retrieved from https://doi.org/10.1145/286942.286947

[12] Department of Computer Science and Engineering. 2013. CSE341: Programming languages.

[13] Scott Danforth and Chris Tomlinson. 1988. Type theories and object-oriented programming. *ACM Computing Surveys, Vol. 20, No. 1* (1988).

[14] Stephen Diehl. Exotic programming ideas: Part 3 (effect systems). Retrieved November 13, 2021 from https://www.stephendiehl.com/posts/exotic03.html

[15] Joe Duffy. 2016. The error model. Retrieved April 15, 2019 from http://joeduffyblog.com/2016/02/07/the-error-model/

[16] Tony Garnock-Jones. 2012. Type inference for ml.

[17] Jonathan Gros-Dubois. 2017. Statically typed vs dynamically typed languages. Retrieved July 27, 2019 from https://hackernoon.com/statically-typed-vs-dynamically-typed-languages-e4778e1ca55

[18] Stefan Hanenberg and others. 2013. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering* (2013).

[19] Robert Harper. 2000. *Type systems for programming languages 1 (draft)*.

School of Computer Science, Carnegie Mellon University.

[20] Robert Harper. 2016. *Practical foundations for programming languages, second edition*. Carnegie Mellon University.

[21] Danny van Heumen. 2016. Error handling in modern languages. Retrieved April 10, 2020 from https://www.dannyvanheumen.nl/post/error-handling-in-modern-languages/

[22] Sir Charles Antony Richard Hoare. The billion dollar mistake. Retrieved November 1, 2021 from https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

[23] Diane Hosfelt. Retrieved October 15, 2021 from https://hacks.mozilla.org/2019/01/fearless-security-memory-safety/

[24] Sujay Jayakar. 2020. Rewriting the heart of our sync engine. Retrieved November 30, 2021 from https://dropbox.tech/infrastructure/rewriting-the-heart-of-our-sync-engine

[25] Walid Taha Jeremy G. Siek. 2006. Gradual typing for functional languages.

[26] Waleed Khan. Null-tracking, or the difference between union and sum types. Retrieved October 17, 2021 from https://blog.waleedkhan.name/union-vs-sum-types/

[27] Vlad Krasnov. 2019. BoringTun, a userspace wireguard implementation in rust. Retrieved November 30, 2021 from https://blog.cloudflare.com/boringtun-userspace-wireguard-rust/

[28] Shriram Krishnamurthi. 2017. *Programming languages: Application and interpretation*.

[29] Angelika Langer. Fundamentals of java generics. Retrieved October 24, 2021 from http://www.angelikalanger.com/GenericsFAQ/FAQSections/Fundamentals.html

[30] Xavier Leroy. 1992. Polymorphic typing of an algorithmic language. PhD thesis.

University Paris VII.

[31] Xavier Leroy. 1998. An overview of types in compilation. In *TIC 1998: Workshop types in compilation* (LNCS), Springer, 1–8. DOI:https://doi.org/10.1007/BFb0055509

[32] Ryan Levick. Why rust for safe systems programming. Retrieved November 29, 2021 from https://msrc-blog.microsoft.com/2019/07/22/why-rust-for-safe-systems-programming/

[33] Ryan Levick. We need a safer systems programming language. Retrieved November 29, 2021 from https://msrc-blog.microsoft.com/2019/07/18/we-need-a-safer-systems-programming-language/

[34] Lars Birkedal Mads Tofte. 1998. A region inference algorithm. *ACM Transactions on Programming Languages and Systems, Vol. 20, No. 5, July 1998, Pages 724–767.* (1998).

[35] Martim Nascimento. 2017. Python duck typing (or automatic interfaces). Retrieved July 29, 2019 from https://martim00.wordpress.com/2017/10/01/python-duck-typing-or-automatic-interfaces/

[36] Oscar Nierstrasz and others. 2005. On the revival of dynamic languages.

[37] James Parmer. 2018. Functional programming: Type systems. Retrieved May 11, 2019 from https://www.youtube.com/watch?v=hy1wjkcIBCU

[38] Benjamin C. Pierce. 2002. *Types and programming languages*.

[39] Aarne Ranta. 2012. *Implementing programming languages*. Retrieved from http://www.grammaticalframework.org/ipl-book/

[40] D; Filkov Ray B; Posnett. 2014. A large scale study of programming languages and code quality in github. In *FSE 2014: Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering*.

[41] Barry Ruzek. 2007. *Effective java exceptions*. Retrieved April 15, 2019 from

https://www.oracle.com/technetwork/java/effective-exceptions-092345.html

[42] James Sinclair. Algebraic data types. Retrieved October 17, 2021 from https://jrsinclair.com/articles/2019/algebraic-data-types-what-i-wish-someone-had-explained-about-functional-programming/

[43] Chris Smith. 2010. Retrieved April 15, 2019 from http://blogs.perl.org/users/ovid/2010/08/what-to-know-before-debating-type-systems.html

[44] Scott F Smith. 2016. *Principles of programming languages*.

[45] svick. 2014. Is it possible to have a dynamically typed language without duck typing? Retrieved April 10, 2020 from https://softwareengineering.stackexchange.com/a/259977/90623

[46] Paweł Święcki. 2018. First steps with python type system. Retrieved July 27, 2017 from https://blog.daftcode.pl/first-steps-with-python-type-system-30e4296722af

[47] Onur Tolga Şehitoğlu. 2008. Programming languages: Type systems.

[48] Coquand Thierry. 2018. Type theory. *The Stanford Encyclopedia of Philosophy (Fall 2018 Edition)*. Retrieved May 7, 2019 from https://plato.stanford.edu/entries/type-theory/

[49] Mads Tofte and Jean Pierre Talpin. 1994. Region based memory management. (1994).

[50] Noel Waghorn. Sum types in swift and kotlin. Retrieved October 17, 2021 from https://adapptor.com.au/blog/sum-types-in-swift-and-kotlin

[51] David Walker and Kevin Watkins. 2001. On regions and linear types. (2001).

[52] Wikipedia. Formal methods. Retrieved October 2, 2021 from https://en.wikipedia.org/wiki/Formal_methods

[53] Wikipedia. Type systems. Retrieved April 15, 2019 from https://en.wikipedia.

org/wiki/Type_system

[54] Wikipedia. Russell's paradox. Retrieved October 10, 2020 from https://en.wikipedia.org/wiki/Russell%27s_paradox

[55] Wikipedia. Curry–Howard correspondence. Retrieved October 10, 2020 from https://en.wikipedia.org/wiki/Curry%E2%80%93Howard_correspondence

[56] Wikipedia. Abstract data type. Retrieved August 5, 2019 from https://en.wikipedia.org/wiki/Abstract_data_type

[57] Wikipedia. Polymorphism. Retrieved April 10, 2020 from https://en.wikipedia.org/wiki/Polymorphism_(computer_science)

[58] Wikipedia. Subtyping. Retrieved April 10, 2020 from https://en.wikipedia.org/wiki/Subtyping

[59] Wikipedia. Covariance_and_contravariance_(computer_science). Retrieved November 6, 2021 from https://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)

[60] Wikipedia. Algebraic data types. Retrieved October 17, 2021 from https://en.wikipedia.org/wiki/Algebraic_data_type

[61] Wikipedia. Tagged union. Retrieved October 17, 2021 from https://en.wikipedia.org/wiki/Tagged_union

[62] Wikipedia. Aliasing. Retrieved November 24, 2021 from https://en.wikipedia.org/wiki/Aliasing_(computing)

[63] 2009. Static vs. dynamic typing of programming languages. Retrieved July 27, 2019 from https://pythonconquerstheuniverse.wordpress.com/2009/10/03/static-vs-dynamic-typing-of-programming-languages/

[64] 2017. The rust programming langage (online book).

[65] Type theory. Retrieved May 7, 2019 from https://ncatlab.org/nlab/show/type+

theory

[66] Soundness and completeness. Retrieved January 1, 2020 from https://flow.org/en/docs/lang/types-and-expressions/

[67] Are unsound type systems wrong? Retrieved December 27, 2019 from https://blog.ambrosebs.com/2018/04/07/unsoundness-in-untyped-types.html

[68] Type compatibility - typescript. Retrieved December 30, 2019 from https://www.typescriptlang.org/docs/handbook/type-compatibility.html

[69] Rust documentation - std. Retrieved November 1, 2021 from https://doc.rust-lang.org/std/result/

[70] Memory management reference. Retrieved November 13, 2021 from https://www.memorymanagement.org/glossary/r.html#term-region-inference

[71] Rust documentation - ownership. Retrieved November 24, 2021 from https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

[72] The rust programming language. Retrieved November 28, 2021 from https://doc.rust-lang.org/book

[73] Linear types make performance more predictable. Retrieved November 13, 2021 from https://www.tweag.io/blog/2017-03-13-linear-types/

[74] Koka. Retrieved November 13, 2021 from https://www.microsoft.com/en-us/research/project/koka/