ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Blaha** Jméno: **Rostislav** Osobní číslo: **500451**

Fakulta/ústav: **Fakulta informačních technologií**

Zadávající katedra/ústav: **Katedra softwarového inženýrství**

Studijní program: **Informatika**

Studijní obor: **Webové a softwarové inženýrství**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Přehled přístupů postupného typování v dynamických programovacích jazycích

Název bakalářské práce anglicky:

An overview of gradual typing approaches in dynamic programming languages

Pokyny pro vypracování:

The lack of explicit type annotations gives dynamic programming languages flexibility, yet the lack of static typing can hinder the maintenance and evolution of the program. Recently we have seen the rise of gradual typing, allowing one to provide type annotations selectively, mixing both typed and untyped modules.
This thesis aims to provide an overview of these approaches in selected programming languages. Next, based on the analysis, the thesis should discuss which techniques could be used for the R programming language.

Seznam doporučené literatury:

Jméno a pracoviště vedoucí(ho) bakalářské práce:

doc. Ing. Filip Křikava, Ph.D. katedra teoretické informatiky FIT

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **29.12.2022** Termín odevzdání bakalářské práce: **11.05.2023**

Platnost zadání bakalářské práce: _____

doc. Ing. Filip Křikava, Ph.D. podpis vedoucí(ho) práce	Ing. Michal Valenta, Ph.D. podpis vedoucí(ho) ústavu/katedry	doc. RNDr. Ing. Marcel Jiřina, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____ _____

Datum převzetí zadání Podpis studenta

Bachelor's thesis

AN OVERVIEW OF GRADUAL TYPING APPROACHES IN DYNAMIC PROGRAMMING LANGUAGES

Rostislav Blaha

Faculty of Information Technology
Department of Software Engineering
Supervisor: doc. Ing. Filip Křikava, Ph.D.
May 11, 2023

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Blaha Rostislav. *An overview of gradual typing approaches in dynamic programming languages.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

List of Figures

List of Tables

List of code listings

I would like to express my gratitude primarily to my thesis supervisor, Filip Křikava, for his guidance, support, and feedback. To Daniel Crha, who provided me with advice and feedback, as well as suggestions on how to distribute my workload. To my family, friends, and colleagues for their support and understanding.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act..

In Prague on May 11, 2023 .....

Abstract

Gradual typing is a feature that allows programming languages to combine dynamic and static typing within the same codebase, enabling the incremental addition of type annotations as the code evolves. This study investigates existing approaches to gradual typing in Python, Ruby, and PHP, with the goal of identifying techniques that could be applied to the R programming language.

The thesis synthesizes information from multiple sources such as documentation, academic articles, blog posts, formal proposals, and forum discussions. It follows up with suggestions on implementation, syntax, semantics, tool support, and adoption strategies for gradual typing in R.

Keywords gradual typing, extending programming languages, R, python, ruby, PHP, dynamic programming languages

Abstrakt

Graduální typování je vlastnost programovacího jazyka, která umožňuje kombinaci dynamického a statického typování ve stejné codebase. Tím umožňuje přidávání typových anotací průběžně s tím, jak se kód postupně rozšiřuje. Cílem této práce je prozkoumat stávající přístupy ke graduálnímu typování v progamovacích jazycích Python, Ruby a PHP, a určit, které techniky by bylo možné aplikovat na programovací jazyk R.

Práce shrnuje informace z vícero zdrojů zahrnujících dokumentaci, odborné články, blogové příspěvky, formální návrhy a diskuse na fórech. Následně přichází s návrhy ohledně implementace, syntaxe, sémantiky, nástrojů a strategií pro přijetí graduálního typování komunitou jazyka R.

Klíčová slova graduální typování, rozšíření programovacích jazyků, R, python, ruby, PHP, dynamické programovací jazyky

Seznam zkratek

RTTI	Real-time Type Information
AST	Abstract Syntax Tree
CFG	Control Flow Graph
PyPI	Python Package Index
OSS	Open Source Software
IDE	Integrated Development Environment
CI	Continuous Integration
DSL	Domain Specific Language
JIT	Just In Time
GUI	Graphical User Interface
OO	Object-oriented

Chapter 1

Introduction

Programming languages are specific in that they are designed to serve communication between humans and machines. For that reason, they need to be unambiguous and executable.

All data in the classical computer is represented by binary digits (bits), which are either zeroes or ones. Different kinds of data, such as integers, floating-point numbers, and characters, require different operations. To enable a system to identify the appropriate operations for a given piece of data, the underlying data type must be stored and passed along with the binary representation. This is where *type systems* in programming languages play a crucial role, helping to encode, decode, and manipulate data correctly while ensuring the correct operations are applied based on the data type.

We can categorize type systems by the type-checking approach used, dividing them into *static* and *dynamic*.

Statically typed languages perform type checks at compile time. These checks aim to prevent code containing errors from running. A typical feature of statically typed languages is mandatory type annotations. As an example, take a look at the statically typed code written in the Scala language. Notice the type `String` separated by a colon from the variable name:

```
// Mutable variable of type String
var myVar: String = "Hello, world!"
println(s"$myVar")
```

Dynamically typed languages, on the other hand, determine the type of variables at runtime. Type checks also happen at runtime, which means that it is generally possible to run code that could lead to errors. While type annotations might be supported, they are generally not required. See the same code written in a dynamically typed language, JavaScript. Notice that the variable type is not explicitly stated anywhere:

```
// Mutable variable of type String
let myVar = "Hello, world!";
console.log(myVar);
```

While the examples mentioned above serve as a good illustration of how statically and dynamically typed code can look like, they might be a bit misleading. Modern statically typed languages

(including Scala) employ type inference algorithms to identify the variable's type without explicitly annotating it. As an example, take a look at the statically typed code with type inference written in the Scala language. Notice that the variable type is not explicitly declared. However, it is still checked during compile time:

```
// Mutable variable of type String
var myVar = "Hello, world!"
println(s"$myVar")
```

Gradual typing, which combines the advantages of both systems, has emerged as a new approach in programming languages [1]. It allows developers to start with dynamically typed code, and as the codebase grows and the need for safer and more robust code arises, they can gradually transition to statically typed code [1]. This means that both statically and dynamically typed code can coexist not just within the same codebase, but even within the same file [1]. This approach has already been successfully used in other languages (JavaScript [2], Python [3], Clojure [4], ...).

The R language serves as a vital computational tool for research across various fields such as statistics, biology, physics, mathematics, chemistry, economics, geology, and medicine [5]. It is an interactive language without explicit types [6]. This allows users to manipulate data structures and functions at runtime [6]. Implicit types are not always consistent or predictable across different operations or functions [6].

The dynamic nature of the language might be beneficial for quick learning and rapid iterations. However, it has negative effects on reusability and can lead to unexpected errors and bugs [6]. This makes it less than optimal, especially for larger code bases and for libraries. A solution to this issue might be gradual typing.

Adding gradual typing to R presents a significant challenge. First, the language itself is highly dynamic, allowing for a plethora of implicit type conversions. The language contains several separate object-oriented systems. It also includes implicit types that are difficult to annotate due to their complexity (most notably, data frames). One of R's strengths is its brevity and interactivity. Both of these qualities should be preserved to ensure that the existing community of users accepts the changes to the language.

This thesis discusses strategies for adding this feature to R based on case studies from three other popular languages – Python, Ruby, and PHP. To the author's best knowledge, this type of analysis has not been done before. The thesis loosely extends the 2019 paper *Towards a Type System for R* [7] and the 2020 paper *Designing Types for R, Empirically* [8] by Turcotte et al. The former paper calls for adding a type system to R and suggests using the gradual typing approach, while the latter proposes a type system for R based on empirical analysis.

1.1 Goals

The main goal of this thesis is to provide an overview and synthesis of the approaches to gradual typing in other languages. It maps the history of adding this feature to three popular languages and discusses the pros and cons of various design decisions. Additionally, it aims to identify opportunities for further research in the area of gradual typing in the R programming language.

The rest of the thesis is organized as follows:

- The *Background* chapter offers definitions of basic terms and an overview of the approach for adding type annotations to dynamic languages. It delves into the advantages of gradual typing as well as various concerns related to implementation, syntax, and semantics.

- The *Review of Gradual Typing in Other Languages* chapter compares the approaches employed in Python, Ruby, and PHP programming languages. These three languages were specifically chosen due to their popularity [9, 10, 11] and their shared characteristic of being dynamically typed, with the option to write statically typed code added later.
- The *Towards Gradually Typed R* chapter introduces the R programming language and discusses its need for gradual typing. Furthermore, it examines design decisions associated with implementing gradual typing in R, informed by insights from the previous sections.
- The *Conclusion* chapter aims to summarize the findings and suggest opportunities for further research in this area.

Chapter 2

Background

This chapter establishes basic terminology associated with types and type systems in general as well as the gradual typing approach. It also initiates the discussion surrounding implementation strategy, syntax, and semantics, which will continue throughout this thesis.

2.1 Types

Bits in memory can represent various forms of data, such as instructions, addresses, characters, integers, and floating-point numbers. On the hardware level, there is no difference between them. To utilize these representations in higher-level programming languages, it is beneficial to store information about the nature of the data. This information is referred to as a *type*. Types serve the purpose of differentiating between various kinds of data and restricting operations to only those that are semantically valid. *Type system* consists of a set of rules linking types and language constructs like variables, expressions or functions and another set of rules for *type equivalence*, *compatibility*, and *inference* [12, p. 298, 299]. In this section, key concepts and terminology of type systems will be discussed.

Type Safety. Type systems are considered *type safe* if they prevent operations that could lead to a violation of the type system's rules (*type error*) from occurring. Type safety can be achieved by examining the compatibility of types involved in an operation. Such checks can be performed during the *compile time*, which occurs before the program is executed, using specialized tools like a *static type checker* or incorporating type checking within a *compiler*. Alternatively, the checks can be conducted during the program execution, also known as *runtime*. It is also possible to do a combination of both [12, p. 299, 300].

Type Soundness. Type systems are considered *type sound* if a program adhering to the rules and semantics of the programming language cannot result in a runtime error [13]. Such property of type system can be proven using a two-part theorem consisting of *progress* and *preservation*. Progress says that if a term passes the type checker it will be able to make the next step of evaluation until fully evaluated. Preservation says that the result of this step will have the same type as the original. This logically leads to the final term having the same type as the original one and the type system is therefore sound [14].

Type Checking. *Type checking* is an essential aspect of programming languages that ensures a program adheres to the language's type compatibility rules. Type checking is an important tool for preventing bugs, improving code maintainability, and enhancing code readability.

There are two main categories of type checking: *statically typed* languages perform type checking at compile time, while *dynamically typed* languages perform type checking at runtime [12, p. 299, 300]. It is important to discuss the trade-offs between these two approaches, such as flexibility versus safety and ease of use versus performance benefits, as they have implications for different programming scenarios.

For statically typed languages, the type checker guarantees that only values of the correct type can be used, eliminating the need for runtime type checking. This leads to both space (saving representation) and time (eliminating runtime checks) performance benefits for programs. The cost to the developer for this benefit is the need to convince the type system that their program does not induce type errors. Due to limitations of decidability, even programs that might have run without error might not be compliant with the type checker [14].

Type Annotations. To generate useful error messages, the type checker needs to know the correct type within the given context. One way to let it know is to add *type annotations*, which are an explicit specification of the expected data type of a variable, function argument, or return value. While type annotations improve both the type checker's and human's understanding of the code, having to write them can be seen as an overhead for the programmer [15].

Type Inference. To spare the programmer this overhead, we can employ a *type inference* algorithm. Such an algorithm is in many cases able to infer the type of many expressions based on their context even without explicit type annotations. A popular choice is the Damas-Hindley-Milner type inference algorithm [16]. The algorithm assigns a type variable to each expression, identifies constraints, turns them into a system of equations, and solves the equations using a method called unification [17]. By using type inference algorithms like Damas-Hindley-Milner, developers can balance the benefits of a statically typed language with a relatively concise and easy-to-write syntax.

Taxonomy of types. To further understand the type systems in programming languages, it is helpful to classify types into different categories. This thesis will follow the taxonomy described by Scott [12, p. 305]:

- *Numeric types*: including floating-point and discrete types;
- *Enumeration types*: a set of named elements, such as booleans or days of the week;
- *Subrange types*: a continuous subset of values of some discrete base type, e.g., `0..100`;
- *Composite types*: non-scalar types, such as container types (arrays, lists, sets, etc.), and unions of types.

Other common types like strings or chars, can depending on the given language and implementation fit either into the Numeric or Composite category.

Type Compatibility. *Type compatibility* refers to determining which type can be used in a specific context. For example, the types of the operands of `+` must both be compatible with some common type that supports addition [12, p. 320]. On the other hand, *type equivalence* involves identifying which types are considered the same. *Type conversion* is the process of creating a new value in a different data type based on an existing value while preserving its original meaning or representation [12, p. 312, 313]. This can happen either implicitly (the compiler injects the call) or explicitly. *Type coercion* is an automatic, implicit conversion in certain contexts [12, p. 321]. For example, let's say a comparison `1==1.0` is performed where `1` is an integer while `1.0` is a float – if the type system supports coercion, such an expression would evaluate to `true`.

Polymorphism. Based on Scott [12, p. 302], *polymorphism* refers to code that is designed to work with values of multiple types. These types must have common characteristics, and the code must rely on no other characteristics. *Subtype polymorphism* occurs when code is designed to work with a specific type, denoted as `T`. Programmers can define other types as *extensions* or

refinements of `T`, and the code works with these *supertypes* or *subtypes* as well. In this context, the subtyping relationship notation `A`$\prec$`B` is used, meaning that `A` is a subtype of `B`. *Parametric polymorphism* refers to code that takes a type or a set of types as a parameter, either explicitly (also known as *generics* or *templates*) or implicitly.

Subtyping can be classified into different categories [12, p. 332]. *Structural typing* is a form of subtyping where names are inessential, and subtyping is defined directly based on the structures of types. Types with the same structure are considered equivalent. In contrast, *nominal typing* considers names to be significant, and subtyping is explicitly defined. Another approach to subtyping is *duck typing*, which is similar to structural typing but applicable to dynamically typed languages. In duck typing, an object is considered to have an acceptable type if it supports the requested methods.

Type bounds are constraints on type parameters in parametric polymorphism, specifying the range of acceptable types. It is possible to specify that a component of a complex type `I` can accept only subtypes or only super types of a type `A`. For example in Scala syntax `class I[T <: A]` tells us that parameter `T` needs to be a subtype of type `A` [18].

Variance describes how the subtyping relationship between more complex types (e.g., container types) relates to subtyping between their components [19, p. 185]. For example, given simple types `A` as a component of a complex type `I<A>` and `B` as a component of a complex type `I<B>`, the following relationships can exist:

- *Covariant*: A relationship where `A`$\prec$`B` implies `I<A>`$\prec$`I<B>`. For example, if `integer` is a subtype of `number`, then a list of type `List<integer>` is a subtype of a list of type `List<number>`.
- *Contravariant*: A relationship where `A`$\prec$`B` implies `I<B>`$\prec$`I<A>`. For example, if `integer` is a subtype of `number`, then a function that takes a parameter of type `I<number>` is a subtype of a function that takes a parameter of type `I<integer>`.
- *Invariant*: The relationship between `A` and `B` does not have any effect on the relationship between `I<B>` and `I<A>`.

In this section, the theory of types, including concepts like type compatibility and polymorphism, was covered. These concepts are essential for understanding the interaction between static and dynamic typing. The next section will show how to bridge the gap between the two, allowing programmers to benefit from the strengths of both approaches.

2.2 Gradual Typing

In their 2006 paper, Siek and Taha [20] reviewed the advantages and disadvantages of both static and dynamic type systems. They argued that dynamic types are better suited for prototyping and scripting, whereas static types are more appropriate for algorithms, data structures, and system programming. The authors noted that it is common for programmers to start with dynamically typed languages and switch to statically typed languages as the program grows. They argued that such a change is unnecessarily costly and that programming languages should provide mechanisms for a smoother transition. Programmer should have control over which parts of code are statically and which are dynamically typed as well as an option to gradually move between the two. To describe such a system, Siek and Taha coined the term *gradual typing*.

2.2.1 Implementation Strategy

There are several approaches to achieving gradual typing. For easier further reference, a name was assigned to each of them:

- *Static Retrofitting*: Adding the ability to write statically typed code to an otherwise dynamically typed language [20].

- *Dynamic Retrofitting*: Adding the ability to write dynamically typed code to an otherwise statically typed language [21].

- *Natively gradual*: Building a gradually typed language from the ground up [22].

This thesis focuses on static retrofitting.

Selecting this approach leaves developers with a decision regarding whether to leave type checking to external tools or modify the compiler or interpreter to perform type checking. The first option allows for the emergence of tools with various approaches, each having different philosophies, pros, and cons, enabling programmers to choose the tool that best suits their needs. External tools avoid performance overhead for partially annotated code, as there is no runtime difference between annotated and unannotated code. However, using external tools might not allow for optimizations that require the interpreter or compiler to know type information. Additionally, it doesn't enable sound gradual typing, as for sound gradual typing to work, some runtime checks on the boundary between the statically and dynamically typed parts of the code are needed in addition to the static ones [23].

Another concern is designing an appropriate type system. Ideally, the syntax of the type system should not conflict with existing language constructs and should be compatible with the language's philosophy and design principles. The ultimate goal is striking a balance between conciseness, readability, and usability on one side, and flexibility that allows developers to precisely annotate various type-related language constructs on the other. The semantics of the type system should be logical, type safe, performant, scalable, and comprehensive, covering as large a subset of valid dynamically typed programs as possible.

2.2.2 Syntax Considerations

When retrofitting static types to a dynamically typed language, developers face several design decisions. One such decision is whether to exploit the existing syntax (e.g., comments) or extend the language's syntax. Exploiting existing syntax can be easier initially since it does not require changes to the core language, but it might conflict with other uses of the exploited language feature and may be less readable. A balanced approach, as demonstrated in Subsection 3.1.3, involves prototyping this feature by exploiting comments and then transitioning to an extended language syntax with type annotations as the design matures.

It is crucial to recognize that existing developer communities of dynamically typed languages are often not used to type annotations. Despite their usefulness, using them might feel cumbersome for those unfamiliar with them. This is why it might be beneficial for the adoption of gradual typing to limit the number of manually added type annotations as much as possible.

One method to reduce reliance on type annotations is by employing a type inference algorithm. As demonstrated by Turcotte et al. [8], type inference can also be used during runtime to annotate large sets of unannotated packages. This approach can serve as a proof of concept or even help with interactions between annotated and unannotated code.

Another factor that might discourage dynamic language programmers from using gradual typing is the complexity of type annotations, especially for compound and polymorphic types. To address this issue, introducing *type aliasing* can be helpful, as it allows developers to map a complex type to a simple type alias, making the annotations more manageable and readable.

Lastly, the number of types included in the type system can also be problematic. To mitigate this issue, employing features such as *union types* and *intersection types* might be beneficial. Union types represent a value that can be one of several types, while intersection types represent a value that needs to satisfy all specified types. These features can help streamline the type system and make it more approachable for developers working with dynamically typed languages.

2.2.3 Semantics Considerations

Introducing gradual typing to a dynamically typed language might bring several challenges, including dealing with existing (implicit) type systems that may be complex and poorly documented [6]. Significant design decisions are needed, such as defining rules for type compatibility, determining which conversions and coercions should be supported, and designing composite types that are concise, useful, and easy to understand. These types should allow for polymorphism to work. Adding gradual typing to a language can introduce runtime overhead on statically typed code [3].

Interactions between statically and dynamically typed code. Determining how interactions between statically and dynamically typed code should be designed is crucial for achieving a seamless integration of gradual typing. Examples of such interactions include function calls, variable access, or class instantiation between statically and dynamically typed parts of code. In a recent paper, Greenman et al. [24] identified several possible *type-enforcement strategies* for restricting interactions between statically and dynamically typed code. These strategies balance trade-offs between type soundness, runtime checks, and performance.

- *Erasure* uses types only for static analysis, with no runtime checks performed. While this approach does not guarantee soundness or ensure that the system moderates all boundaries between statically and dynamically typed code, it does provide some benefits. Error messages contain only boundaries relevant to the program, but they may not provide all the necessary information. The erasure strategy accepts most programs without raising runtime errors and does not require wrappers, making it faster compared to other strategies.

- *Transient* checks runtime type only when necessary for an operation to be executed. This strategy allows for type soundness but does not ensure that the system moderates all boundaries between statically and dynamically typed code. Additionally, error messages may not contain all relevant information or only the relevant boundaries. The transient approach accepts fewer programs without raising runtime errors than erasure but more than the natural strategy, and it does not require wrappers, thus offering faster performance.

- *Natural* employs higher-order checks to ensure the integrity of types throughout the entire program. The natural approach allows for soundness and ensures that the system moderates all boundaries between statically and dynamically typed code. Furthermore, error messages contain only the relevant boundaries and provide all necessary information. However, this strategy accepts the fewest programs without raising runtime errors. Unlike the other strategies, it does require wrappers, making it slower in terms of performance.

- *Concrete* mandates descriptive type information as part of the metadata associated with every value in both the statically and dynamically typed parts of the code. Type checks are performed every time a value is used in an operation, providing a more thorough enforcement of type information.

- *Hybrid* combines multiple strategies to achieve a balance between soundness, performance, and flexibility in handling interactions between statically and dynamically typed code.

Understanding the trade-offs between these type-enforcement strategies is essential for selecting the most appropriate approach when integrating gradual typing into a language or system. Each strategy offers unique benefits and limitations, depending on the specific requirements and goals of the implementation.

Sound gradual typing. Realizing type soundness for a gradual type system requires some runtime checks for potential type mismatches between statically and dynamically typed code. Takikawa et al. [23] mention two strategies for reducing the frequency of runtime checks for

potential type mismatches between statically and dynamically code: *macro-level* and *micro-level* gradual typing. Macro-level gradual typing enforces the programmer to always either annotate the whole module or leave it dynamically typed. Micro-level gradual typing assigns an implicit dynamic type to all annotated parts of the program. The implementation inserts casts at the appropriate parts of the code.

Takikawa et al. [23] investigated the performance effects of sound gradual typing by testing various combinations of typed and untyped modules for a set of 12 programs in Typed Racket. They found that out of the 12 fully statically typed programs, 6 showed a speed improvement of up to 72% compared to their dynamically typed counterparts, while one had no effect, and the rest were slower by up to 13.34 times. In theory, fully statically typed programs should always be faster due to compiler optimization. However, in practice, there tend to be dependencies on external unannotated code that needs to be resolved during runtime, which can negatively impact performance.

For partially typed programs, performance slowdowns were observed in all cases, with a mean overhead of up to 53.1 times and a maximum overhead of up to 121.51 times, depending on the program. It is important to note that the study's findings are specific to macro-level gradual typing in Typed Racket, and further research is needed to draw broader conclusions about other approaches [23].

Despite the potential performance cost, sound gradual typing still has many advantages. The additional runtime checks can help better locate type errors. If runtime performance is critical, gradual typing can still be an option by performing static type analysis only. While this does not guarantee soundness, it can still identify a wide range of type errors. In either case, the type annotations can provide valuable information about the code and help improve its maintainability and understandability.

Chapter 3

Review of Gradual Typing in Other Languages

This chapter examines three popular programming languages—Python, Ruby, and PHP—that have already implemented gradual typing in various forms. Each language approaches gradual typing differently and has made a unique set of design decisions from which the R programming language can learn.

3.1 Python

Python is an open-source language with a wide range of applications, including web development, scientific and numeric computing, education, desktop GUIs, software development, and business applications [25]. It is a dynamically-typed language with a simple yet effective approach to object-oriented programming [26, p. 1]. Although its authors take pride in its simplicity and brevity, Python supports advanced features, such as a wide variety of data structures, including lists, tuples, sequences, sets, and dictionaries [26, p. 33–42]. A key design principle of Python is duck typing (see Section 2.1) [26, p. 112].

Created by Guido van Rossum in 1989 at the Stichting Mathematisch Centrum in the Netherlands [27], Python was designed as a successor to the ABC programming language [28, p. 147]. The first internal release occurred in 1990, followed by the first public release, version 0.9.0, in 1991. Python continued to evolve, with version 1.0.0 released in 1994 and version 2.0 in 2000 [27].

Guido van Rossum, who remains the main contributor to Python, continued his work on the language at the Corporation for National Research Initiatives in Reston, Virginia, in 1995, and later at BeOpen.com in 2000 [28, p. 147]. In 2001, the *Python Software Foundation* was formed as a non-profit organization to manage Python-related Intellectual Property [28, p. 147]. Python 3.0 was released in 2008 [27].

Today, Python is one of the most popular and widely used programming languages. It ranks second on GitHub's list of most-used programming languages in 2022 [9], fourth in Stack Overflow's 2022 Developer Survey in the programming, scripting, and markup languages category [10], and first on the TIOBE Programming Community index for April 2023 [11].

3.1.1 Python Built-in Types

While Python is a dynamically typed language, its interpreter works with a range of built-in types [29, p. 31–91]. The principal built-in types are numerics, sequences, mappings, classes,

instances, and exceptions. In this text, a subset of the types will be briefly introduced. For a more comprehensive overview, please see the standard library [29, p. 31–91]. The types relevant to gradual typing will be further elaborated on in the subsequent text.

Numeric Types. Numeric types in Python consist of `int`, `float`, and `complex`. The `int` type represents integer values with unlimited precision. The `float` type is a floating point number. In addition to the built-in numeric types, the standard library includes `fractions.Fraction` for representing rational numbers and `decimal.Decimal` for representing floating-point numbers with user-definable precision. When performing arithmetic operations with mixed numeric types, the "narrower" type operand is coerced to match the wider one. In this context, an integer is considered narrower than a floating-point number, which in turn is narrower than a complex number [29, p. 32–38].

Enumeration Types. Enumeration types in Python include `bool`, which is a subclass of `int`. The `bool` type conceptually represents an enumeration of two values, with `True` being represented by `1` and `False` being represented by `0`. In addition to the built-in `bool` type, the standard library provides the `enum.Enum` base class, which allows users to define their own enumeration types.

Subrange Types. Subrange types in Python include the `range` type, which represents an immutable sequence of numbers. The `range` type is particularly useful for looping a specific number of times in `for` loops.

Composite Types. Composite types in Python encompass a variety of data structures, including:

- `str`: A sequence of Unicode characters.
- `list`: A mutable sequence typically used to store collections of homogeneous items.
- `tuple`: An immutable sequence typically used to store collections of heterogeneous items.
- `set`: An unordered mutable collection of distinct objects, which can be heterogeneous.
- `frozenset`: An unordered immutable collection of distinct objects, which can also be heterogeneous.
- `dict`: A mapping of hashable values to arbitrary objects, allowing for heterogeneous collections.
- `Union`: A type that enables grouping several other types into their common supertype.

According to the Python Standard Library [26, p. 69–81], classes are also considered built-in types. In Scott's taxonomy, they would be categorized as composite types. The Listing 1 demonstrates how the object-oriented system works in Python.

As this is the first Python code snippet in this thesis, please note that indentation has a semantic meaning. The first two lines define a new class named `A`, with an attribute on line 2. Lines 6–9 demonstrate how to add a method to a class after instantiation, showcasing that Python classes behave like hash maps where keys are attribute and method names, and values are their corresponding values or functions. Line 9 shows a method call. Lines 11–22 illustrate how Python supports simple inheritance, while lines 24–32 showcase multiple inheritance.

Lines 34–36 define a new class `C`, which has its own implementation of the `greetA()` method. The `say_hi()` function (lines 38–39) accepts an object and calls its `greetA()` method, demonstrating duck typing in action. Finally, lines 43–44 show the `say_hi()` function working with instances of classes `A` and `C`, regardless of their actual types.

```
class A:
    AttrA = 'Hello'

a = A()

def greetA(self):
    print(self.AttrA + ' World!')

A.greetA = greetA

a.greetA()  # 'Hello World'

class B:
    AttrB = 'Bonjour'

class SubB(B):
    AttrSubB = ' Le Monde!'
    def greetB(self):
        print(self.AttrB + self.AttrSubB)

subB = SubB()
subB.greetB()  # 'Bonjour Le Monde!'

class SubAB(A, B):
    def greetA(self):
        print(self.AttrA + ' World!')
    def greetB(self):
        print(self.AttrB + ' Le Monde!')

subAB = SubAB()
subAB.greetA()  # 'Hello World'
subAB.greetB()  # 'Bonjour Le Monde!'

class C:
    def greetA(self):
        print('Gutten Tag!')

def say_hi(x):
    x.greetA()

c = C()

say_hi(a) # 'Hello World'
say_hi(c) # 'Gutten Tag!'
```

Code listing 1 Demonstration of Python object-oriented system

Having established a solid understanding of Python types, it is time to delve into the evolution and implementation of gradual typing within the language, shedding light on its historical context and development process.

3.1.2 Towards Gradual Typing

The following text delves into the history of gradual typing in Python, exploring the evolution of type annotations and the gradual typing system in the language. The journey of gradual typing in Python is a testament to the community's dedication to improving the language's capabilities, as well as its adaptability to meet the changing needs of its users. We will discuss key milestones and decisions that have shaped Python's gradual typing landscape and set the stage for its current implementation.

In 2004, van Rossum introduced the idea of adding optional static typing to the dynamically typed language in a blog post [30]. He argued that such a change would enable developers to contain bugs faster, thanks to compiler hints about potential issues. However, he also recognized that this change would be difficult to implement and that creating a formal proposal for it would present significant challenges.

Van Rossum also discussed several implementation challenges, such as whether inheritance of types (e.g., $int \prec long \prec float \prec complex$) should be implemented and how it would work with duck typing, what container types should look like, and whether type checking should happen during compile time or runtime [30].

As can be seen from the discussion forum under the blog post [31] the response of the community was generally skeptical. While some commentators were in favor of the change, others strongly opposed it, citing the complexity of both implementing and using static types, the potential distraction from other important issues, and the change defying Python's philosophy of simplicity and minimalism. Some argued that most, if not all, of the expected benefits, could already be achieved with the existing syntax and libraries. Others proposed alternative solutions like using type inference or adding a possibility to run type checks as well as other types of validations before and after method execution (also known as *Design by Contract*).

Van Rossum followed up with another blog post [32] reacting to the critique. He argued that creators of frameworks and large applications need type annotation. This leads to them creating their own solutions and ultimately to a lower level of readability and interoperability of code in the Python community. Other newly listed arguments for adding annotations were documentation, runtime inspection, and refactoring. He noted that a way to optimize code might be using type inference. For it to properly work, some sort of type hints would still be needed as *"Python is so dynamic that worst-case assumptions often make optimizations nearly impossible"*.

He further outlined several ideas for implementing gradual typing to Python [32]. In his view, a type should be an abstract set of method signatures. He suggested that *interface types* should be structural, meaning any type implementing a set of methods defined by an interface should be acceptable. Interfaces should be declared using their own keyword, rather than the existing `class` keyword. He also suggested the possibility of, when needed, annotating an argument nominally. For example, `def foo(x: class int) -> int:` would enforce only the built-in `int` class, as opposed to any class of the same structure (including user-defined).

He proposed a built-in list of standard interfaces, including existing proto-interfaces such as number, file-like, callable, and mapping. The `any` type would be the union of all possible types, while `nothing` would be the union of no types. He also proposed support for *parametrized types*, using a syntax like `class List(list)[T]`. Such syntax could be prototyped using a metaclass.

Subtyping should be part of the type system. He also suggested supporting *union types* using the `|` operator, for example, `def read(f: file | str) -> str`. Lastly, van Rossum discussed several possible syntax options for the format of the signature of functions and methods, such as `x: (int, int) -> str`.

As with the previous blog post, the community's response [33] seemed to be rather lukewarm. Various opinions emerged from the discussions:

- Some welcomed and expanded on the concepts, seeing potential benefits for code readability, maintenance, and performance;
- Several were open towards type annotations but strongly against built-in type checking, which felt unnecessarily complex to them. They argued that the existing dynamic typing system was sufficient and that introducing static typing could lead to increased complexity and a steeper learning curve;
- The remainder was against any form of type annotations, seeing it as against Python's philosophy of simplicity, minimalism, and readability. They believed that adding type annotations would make Python less accessible to new learners and could potentially alienate users who appreciate the language's simplicity [33].

The following comments illustrate some of the more critical responses. Phillip J. Eby, Python contributor and author of books *Web Component Development with Zope 3* and *You, Version 2.0*, commented, "*..., relatively few languages support all the bells and whistles you've got here, and Python certainly has gotten on well enough without most of them.*" Bruce Eckel, blogger and author of many popular books on programming including *Thinking in Java* and *Thinking in C++*, argued, "*I cannot see how traditional static type checking could be applied to Python and have anything like Python come out the other end.*" Ka-Ping Yee, author of a Python course at UC Berkeley, expressed surprise, "*I'm rather surprised that you're suggesting such an ambitious and complex type system. Not that this is a bad thing, but it feels a little out of character for Python as I'm used to it.*" Marek Baczyński claimed, *"Making a type system like what is described in the article would make Python a poor man's ML, because most people will use a feature for the sake of it."* Finally, Rich Salz, a long-time member of the IETF, noted, "*Holy crap. Python has become very successful with the current type system. And you are now considering doubling the complexity of the language.*"

In his third and final blog post [34] on the topic, van Rossum scaled down his proposal which helped to mitigate some of the criticism. He reinforced the originally proposed syntax for parameter and return type declarations for functions:

```
def foo(x: t1, y: t2) -> t3:
    ...body...
```

Until the parser is ready for such syntax, he proposed using decorators – a feature of Python denoted by the leading `@` symbol that allows programmers to modify or extend the behavior of functions or classes by wrapping them in another function. [34]:

```
@arguments(t1, t2)
@returns(t3)
def foo(x, y):
    ...body...
```

He also reiterated the idea of Interfaces with the following syntax [34]:

```
 @interface I1(I2, I3):
    def foo(a: t1, b: t2) -> t3:
        "docstring"

class C(I1): # implements I1
    def foo(a, b):
        return a+b
```

He seemed less confident in typed class attribute declarations and design by contract. Other ideas, including overloaded methods, parametrized types, variable declarations, `where` clauses, union types, cartesian products and compile–time type checking were all declared out of scope for the time being [34].

This updated proposal received less pushback compared to the previous one which can be again illustrated by the adjacent discussion thread [35]. Phillip J. Eby, who previously questioned the necessity of many features, simply expressed his approval with a *"+1."* Bruce Eckel, who had previously argued that traditional static type checking could not be applied to Python without significantly altering the language, did not reiterate his previous concerns in this discussion. Instead, he brought a new take on how tests and test coverage could be implemented. Ka–Ping Yee, who had been surprised by the ambitious and complex type system initially suggested, now expressed support for the updated proposal, stating, *"I like this proposal a lot. The concepts make a lot of sense to me."* Other previous critics, such as Marek Baczyński and Rich Salz, did not weigh into this discussion.

3.1.3 Evolution of Python's Typing System

The development of Python takes place openly. The dev team communicates with each other through forums, blogs, and mailing lists. Once an idea is polished enough, it is turned into a formal proposal or PEP (Python Enhancement Proposal). The PEPs are stored and tracked on the official Python website, providing a unique insight into the actions, motivations, and reasoning of the language's creators [36].

Figure 3.1 displays the history of notable typing PEPs mapped to their year of creation and the version of Python in which they were implemented. The diagram aims to cover all major changes and additions to syntax. For the sake of readability, informational proposals, and proposals focused mainly on performance improvements have been left out. Both types of omitted PEPs are still covered in the text below.

In 2005, Collin Winter introduced *typecheck*, a runtime type-checking module for Python, which used decorators to work [37]. Annotated Python code using this package would look like this [38]:

```
@accepts(Number, Number)
@returns(Number)
def my_add(a, b):
    return a + b
```

In March 2006, Bill Birch started a new blog calling for structural subtyping in general, without focusing on any particular language [39]. In April 2006, van Rossum posted a blog post outlining the scope of Python 3, which did not mention gradual typing or type annotations [40]. However, Birch sought encouragement for contributing to a PEP focused on types in *Python-3000*, the mailing list dedicated to the new version of Python [41]. One month later, he outlined a draft PEP titled "Some Thoughts on Types for Python-3000" that vouched for structural subtyping [42].

While there is no evidence to confirm that Birch's draft was submitted, it had some influence. Collin Winter had questions about Python 3 and its interactions with typecheck and extensively referred to Birch's draft in his post, likely expecting it to be implemented [43]. Just a week later, Winter stated that he was working on notes for the type annotations PEP, implying he was now responsible for its development [44]. He continued discussing his ideas on Python–3000 and refined them in discussions with van Rossum as well as other contributors [45, 46, 47, 48].

PEP 3107 was introduced in December of that year, bringing gradual typing to Python for the first time. The proposal provided syntax only, without any associated semantics. It covered adding type annotations for parameters and return values of regular functions, intentionally

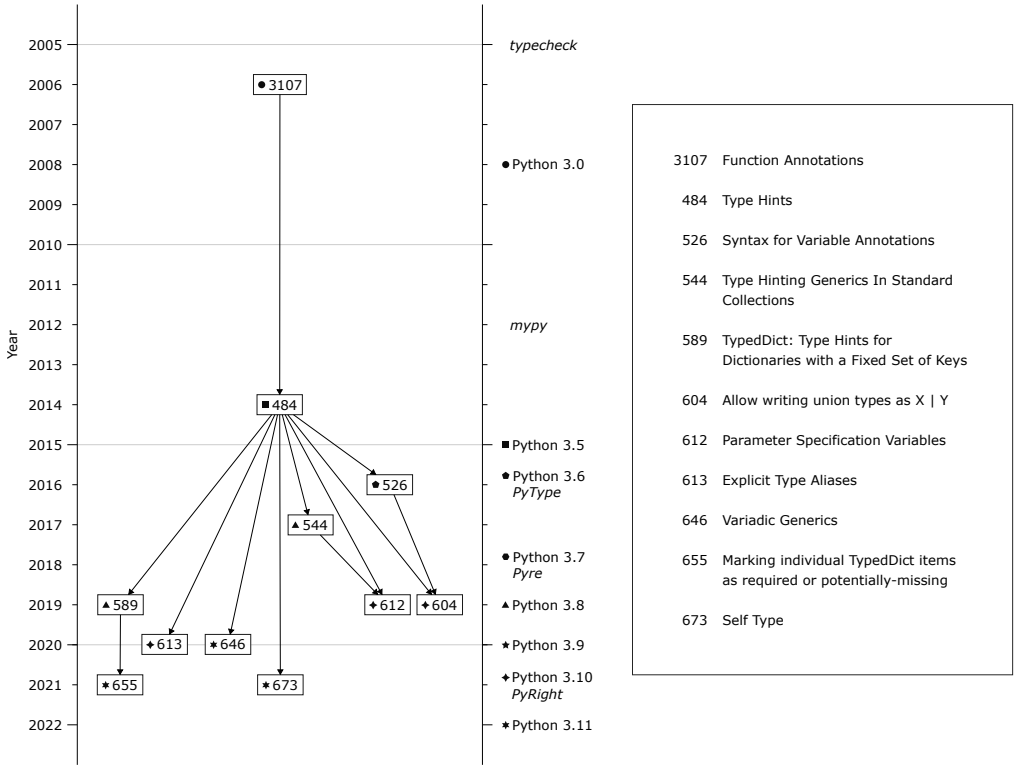

Figure 3.1 History of notable typing PEPs

leaving out other type parameters, including lambda functions [49]. From the outset, the PEP emphasized that this feature was "completely optional," addressing community concerns and the controversy surrounding the topic. It also clarified that there were no plans to add type–checking directly to Python and that external libraries should be used for this purpose [49].

The syntax for type annotations would be:

```
def foo(parameter1: annotation1, parameter2: annotation2) -> annotation3:
    ...
```

Once compiled, the annotations are saved as the `annotations` attribute of a function in the following format [49]:

```
{
    'parameter1': 'annotation1',
    'parameter2': 'annotation2',
    'return': 'annotation3',
}
```

Here, `return` is a reserved word and cannot be used as an attribute name. Type annotations do not have to match either Python's implicit types or any other limited list of values [49].

In 2012, *mypy* a static type checker for Python was first introduced (see Subsection 3.1.6).

The idea of type annotations was expanded in 2014, in *PEP 484* [50], which introduced the first draft of the semantics. The proposal was co–authored by van Rossum. As shown in Figure 3.1, this PEP is influential as it has been cited by many other notable typing PEPs.

The proposal was inspired by mypy and listed support for static analysis as its most important goal. A significant contribution of this proposal is the introduction of the *typing* module, which provided a namespace for new type–related features.

The proposal also included the concept of type aliases. Type aliases allow a programmer to define alternative names for existing types. Here is an example of a simple type alias [50]:

```
Url = str

def retry(url: Url, retry_count: int) -> None:
    ...
```

A more complex example would look like this [50]:

```
T = TypeVar('T', int, float, complex)
Vector = Iterable[Tuple[T, T]]
```

On the first line, a new type variable `T` is defined in a way that allows it to be either int, float, or complex. On the second line, a complex type alias named `Vector` is defined as a tuple of two elements of the same type `T`. Here is an example of using the newly defined type alias [50]:

```
def inproduct(v: Vector[T]) -> T:
    return sum(x*y for x, y in v)
def dilate(v: Vector[T], scale: T) -> Vector[T]:
    return ((x * scale, y * scale) for x, y in v)
vec = [] # type: Vector[float]
```

The `inproduct` function computes the sum of the products of the tuple elements, while the `dilate` function scales the vector by a factor of type `T`. The very last line shows how to declare an empty vector.

Another addition was generics allowing parametrization of containers and functions [50]:

```
T = TypeVar('T')

def isfirst(l: Sequence[T], c: T) -> bool:
    return l[0] == c
```

Here, the type of `c` is not explicitly specified, but it must always be consistent with the type of the `Sequence`. This allows using this method for work with sequences of various type. The PEP clarifies, that consistent in this case means invariant. It is however possible to change this behavior to either covariant or contravariant as needed [50]:

```
# Covariant type
CovT = TypeVar('CovT', covariant=True)

# Contravariant type
ConT = TypeVar('ConT', contravariant=True)
```

It is possible to define a supertype of a set of types using a factory called `Union` [50]:

```
def handle_employees(e: Union[Employee, Sequence[Employee]]) -> None:
    if isinstance(e, Employee):
        e = [e]
    ...
```

Generic classes would be defined like this [50]:

```
T = TypeVar('T')

class LoggedVar(Generic[T]):

    def __init__(self, value: T, name: str, logger: Logger) -> None:
        self.name = name
        self.logger = logger
        self.value = value

    def set(self, new: T) -> None:
        self.log('Set ' + repr(self.value))
        self.value = new

    def get(self) -> T:
        self.log('Get ' + repr(self.value))
        return self.value

    def log(self, message: str) -> None:
        self.logger.info('{}: {}'.format(self.name, message))
```

As there was still no native support for type annotations of variable declaration, programmers were recommended to use comments [50]:

```
x = []                 # type: List[Employee]
x, y, z = [], [], []   # type: List[int], List[int], List[str]
x, y, z = [], [], []   # type: (List[int], List[int], List[str])
a, b, *c = range(5)    # type: float, float, List[float]
x = [1, 2]             # type: List[int]
```

Lastly, this proposal introduced function overloading using the following syntax [50]:

```
class bytes:
    ...
    @overload
    def __getitem__(self, i: int) -> int: ...
    @overload
    def __getitem__(self, s: slice) -> bytes: ...
```

PEP 484 had two adjacent informational PEPs – *PEP 482* created in 2015 and *PEP 483* created in 2014.

PEP483 laid out the theory of the new type hinting proposal. It set the stage by defining a type as a set of values and a set of functions that one can apply to these values [51]. It defined terms like subtyping relationship and gradual typing in a manner consistent with Section 2.2, citing the same sources. The relationship between types and classes was further explored – every class is a type, but not every type is a class. An example of a type that is not a class might be `Union[str, int]`.

The fundamental building blocks of the type system were defined as follows [51]:

- **Any**: A type that can represent any value.
- **Union**: A type that represents a set of types (e.g., `Union[str, int]`).

- **Optional**: A shorthand for a Union that includes `None` (e.g., `Optional[type1]` is equivalent to `Union[type1, None]`).
- **Tuple**: A type that represents a tuple with a fixed number of elements and their respective types (e.g., `Tuple[int, float]` represents a tuple with an `int` as the first element and a `float` as the second).
- **Callable**: A type that represents a function, specifying its input parameter types and return type (e.g., `Callable[[int, float], str]` describes a function that takes an `int` as the first parameter, a `float` as the second, and returns a `str`).

Furthermore, the proposal covered generics, type variables, and variance, concepts that have been discussed earlier in this subsection.

PEP 482 provided a comprehensive literature overview of related work in the field of type systems [52]. The proposal covered existing approaches in Python and their corresponding tools, as well as relevant solutions in other programming languages. The existing tools in Python included:

- mypy: A static type checker that served as an inspiration for PEP 484.
- *Reticulated Python*: An experimental gradual typing system for Python.
- *PyCharm*: An integrated development environment (IDE) that supports type inference and type checking.
- Other: pyflakes, pylint, numpy, Argument Clinic, pytypedecl, numba, and obiwan.

The proposal covered several other programming languages that have developed their type systems to support type hinting and gradual typing, including *ActionScript*, *Dart*, *Hack* (Subsection 3.3.3) and *TypeScript*.

The proposal served as a comprehensive review of the existing landscape of type systems. It also covered both Python-specific approaches and those implemented in other programming languages.

The PEP 484 (and thus concepts covered in PEP 482 and PEP 483) was approved and integrated into version 3.5 released in 2015.

PEP 526 [53], created in 2016, further expanded the ideas on type annotations by introducing syntax for variable declarations, such as `variableName: int` and `containerName: List[int]`. The reasons for this addition included:

- To allow for annotating uninitialized variables (e.g., `a = None # type: int` vs. `a: int`).
- Combining comment annotations and regular comments might cause confusion.
- Retrieving comment annotations during runtime was problematic.

This proposal was approved and incorporated into Python 3.6 in 2016.

3.1.4 Optimizing Python's Typing System

By 2017, it was clear that PEP 484 and PEP 526 were successful, with the typing module being downloaded one million times per month [54]. However, the development was intentionally kept outside the core of the main interpreter, CPython, which led to performance issues. Generic classes, in particular, were *"very slow"*.

PEP 560 [54] aimed to improve performance by removing a series of hacks and bugs. As a result, the following performance improvements were achieved:

- Reloading the typing module became up to 7 times faster.
- The creation of generic classes became up to 4 times faster.
- Generic type operations became 10–20% faster.

Another issue with type annotations as defined by PEP 484 and PEP 526 was forward references, where type hints contained names that were yet to be defined [55]. The workaround at the time was to use string literals (instead of `variable: type`, programmers would use `variable: 'type'`). *PEP 563* [55] addressed this issue by postponing the evaluation of annotations, which also reduced the runtime cost of annotations.

PEP 561 [56] established a standardized way to distribute packages with type information (including the type definitions). Type information was distributed in the form of *stubs* or non-runnable code stored as `.pyi` files. Stubs were bundled into *distributions* or packaged files ready for release. A distribution could consist of one or more *packages* or directories that namespace modules. To include the type definitions, package maintainers needed to add a "py.typed" file in the top-level directory.

All PEP 560, PEP 561, and PEP 563 were approved and released as part of Python 3.7 in 2018.

PEP 544 [57] created in 2017, proposed the addition of support for structural subtyping. Given Python's familiarity with the concept of duck typing, the proposal dubs structural subtyping "Static Duck Typing." The proposal introduces the term "protocols" for describing types that support structural subtyping. The proposed syntax is as follows:

```
from typing import Protocol

class SupportsClose(Protocol):
    def close(self) -> None:
        ...
```

PEP 544 made several classes in the `typing` module into protocols, such as Callable, Awaitable, Iterable, Iterator, AsyncIterable, AsyncIterator, Hashable, Sized, Container, Collection, Reversible, ContextManager, AsyncContextManager, SupportsAbs, and other Supports* classes.

PEP 589, created in 2019, extended the capabilities of type annotations for dictionaries [58]. Dictionaries in Python are data structures consisting of unique "key: value" pairs [26, p. 39]. PEP 484 allowed type annotations for consistent dictionaries only.

PEP 589 introduced the TypedDict feature, which enables more granular type annotations for dictionaries:

```
from typing import TypedDict

class Movie(TypedDict):
    name: str
    year: int

movie: Movie = {'name': 'Blade Runner',
                'year': 1982}
```

This allowed type checkers to better validate dictionary structures by specifying the expected types for each key and their corresponding values. Both *PEP 544* and *PEP 589* were approved and released as part of Python 3.8 in 2019.

Next in the line was *PEP 585* [59]. Due to the historical separation between the typing module and Python core, the language ended up with a duplicated collection hierarchy. This was initially needed to enable support for generics. The proposal resolved this issue, making the

language easier to use and maintain. PEP 585 was approved and released as a part of Python 3.9 in 2020.

Python 3.10 released in 2021 saw the implementation of three important typing PEPs. *PEP 604* [60] (2019), inspired by the Scala programming language, introduced a more concise syntax for union types. Instead of using `Union[X,Y]`, it was possible to simply use `X|Y`. *PEP 612* [61] (2019) improved the ease of annotating higher-order functions and functions with complex signatures. *PEP 613* [62] (2020) introduced the `TypeAlias` type, which explicitly denoted type aliases and helped avoid potential confusion (e.g., changing from `x = int` to `x: TypeAlias = int`).

The following PEPs were implemented in Python 3.11: *PEP 646* [63] (2020) introduced variadic generics, allowing for more precise types for functions and classes that handle an arbitrary number of arguments. This improved type-checking and readability in Python code. *PEP 655* [64] (2021) extended PEP 589 by providing native support for optional keys in TypedDict, which could previously only be achieved through workarounds. With this PEP, qualifiers `typing.Required` and `typing.NotRequired` could be used:

```
class Movie(TypedDict):
    title: str
    year: NotRequired[int]
```

PEP 673 [65] (2021) introduced type `Self` as a simple syntax for methods returning an instance of their class:

```
from typing import Self

class Shape:
    def set_scale(self, scale: float) -> Self:
    self.scale = scale
    return self

class Circle(Shape):
    def set_radius(self, radius: float) -> Self:
    self.radius = radius
    return self
```

Python 3.11 was released in 2022 and is the latest stable version as of the day of this writing.

3.1.5 Typed Python in 2023

Throughout this section, it has been shown how Python has come a long way in implementing gradual typing. The current version supports both statically and dynamically typed code without an obvious bias towards either of them. Recent research has shown that type annotations are widely used and are effective in exposing type errors. However, they are not always used consistently, and it seems that not all users routinely employ a type checker [66].

The Listing 2 demonstrates how the same code can be written in Python both with and without annotations.

```python
# Untyped code
class Vec2D:
  def __init__(self, x, y):
    self.x, self.y = x, y

  def __add__(self, other):

    if isinstance(other, (int, float)):
      other = Vec2D(other, other)

    return Vec2D(
      self.x + other.x,
      self.y + other.y
    )

  def __str__(self):
    return f"({self.x}, {self.y})"

vec1 = Vec2D(1, 2)
vec2 = Vec2D(3, 4)
vec3 = Vec2D(1.5, 2.5)

print(vec1 + vec2)  # Output: (4, 6)
print(vec1 + vec3)  # Output: (2.5, 4.5)
print(vec1 + 2)     # Output: (3, 4)
```

```python
# Typed code

from typing import Generic, TypeVar, Union

T = TypeVar("T", int, float)

class Vec2D(Generic[T]):
  def __init__(self, x: T, y: T) -> None:
    self.x, self.y = x, y

  def __add__(self,
    other: Union['Vec2D[T]', T]) -> 'Vec2D[T]':
      if isinstance(other, (int, float)):
        other = Vec2D(other, other)

      return Vec2D(
        self.x + other.x,
        self.y + other.y
      )

  def __str__(self) -> str:
      return f"({self.x}, {self.y})"

vec1 = Vec2D(1, 2)
vec2 = Vec2D(3, 4)
vec3 = Vec2D(1.5, 2.5)

print(vec1 + vec2)  # Output: (4, 6)
print(vec1 + vec3)  # Output: (2.5, 4.5)
print(vec1 + 2)     # Output: (3, 4)
```

Code listing 2 Comparison of Python code without and with type annotations

In the typed version (to the right), type annotations provide more information about the expected input and output types for methods and functions, enhancing code readability and enabling type checking:

- A `TypeVar`, defined on line 3 of the typed code, is used to represent either an `int` or a `float`. This `TypeVar` is further employed on lines 7, 8, and 12 to demonstrate the use of generic types within the `Vec2D` class.
- The constructor on line 7 includes type annotations for both input parameters and the return type (or the absence thereof). This makes it clear that the constructor accepts two arguments of either `int` or `float` type and returns nothing.
- The `add` method, starting on line 11, showcases the use of Union types and forward references. The type annotation for the right operand of the addition operator, specified on line 12, demonstrates that it can be either an instance of the `Vec2D` class or a scalar value (`int` or `float`).
- The forward reference to the `Vec2D` class (line 12 of the typed code) is used to indicate the return type of the `add` method, which is an instance of `Vec2D`.

- The `add` method, on lines 13–18, performs different operations depending on the type of input. If the other argument is an instance of `Vec2D`, it performs vector addition. In the case of the scalars it first converts them into a `Vec2D` instance by setting both the x and y components to the scalar value, and then performs the addition.

Pandas. Pandas is a powerful data analysis library for Python, providing a `DataFrame` object for data manipulation with integrated indexing [67]. It is built on top of an array programming library called *NumPy* [68, 67]. `DataFrame` is a type that essentially represents a table with named rows and columns. This makes it especially relevant to R, which offers similar functionality.

```
import pandas as pd
from typing import Tuple, TypeAlias

def create_dataframe(
    data:
        Tuple[
            Tuple[int, str, int],
            Tuple[str, int, str],
            Tuple[int, str, int]
        ]
) -> pd.DataFrame:
    return pd.DataFrame(
        data,
        index=['row1', 'row2', 'row3'],
        columns=['col1', 'col2', 'col3']
    )

df = create_dataframe((
    (1, "a", 3),
    ("b", 5, "c"),
    (7, "d", 9)
))

print(df)
print(df.dtypes)

# Python Console
#      col1 col2 col3
# row1    1    a    3
# row2    b    5    c
# row3    7    d    9
# col1    object
# col2    object
# col3    object
# dtype: object
```

In the example above, a function called `create_dataframe` is defined that accepts a tuple of tuples as an argument. The data represents a 3x3 matrix with heterogeneous elements. The function returns a pandas `DataFrame` constructed with the given data, named rows (`'row1'`, `'row2'`, `'row3'`), and named columns (`'col1'`, `'col2'`, `'col3'`). The `DataFrame` is printed along with its data types using the `dtypes` attribute.

The function annotations are somewhat difficult to read, and the output of the `dtypes` function is not particularly useful for this specific case. However, it can be improved using Python functionality alone:

```
import pandas as pd
from typing import List, Tuple, TypeAlias

DataTuple: TypeAlias = Tuple[
    Tuple[int, str, int],
    Tuple[str, int, str],
    Tuple[int, str, int]
]

def create_dataframe(data: DataTuple) -> pd.DataFrame:
    return pd.DataFrame(
        data,
        index=['row1', 'row2', 'row3'],
        columns=['col1', 'col2', 'col3']
    )

df = create_dataframe((
    (1, "a", 3),
    ("b", 5, "c"),
    (7, "d", 9)
))

def get_type(value):
    return type(value).__name__

type_df = df.applymap(get_type)

print(df)
print(type_df)

# Python Console
#      col1 col2 col3
# row1    1    a    3
# row2    b    5    c
# row3    7    d    9
#      col1 col2 col3
# row1  int  str  int
# row2  str  int  str
# row3  int  str  int
```

Using a type alias makes our type annotation more concise and easier to use. With the custom printout, programmers have a better understanding of the `DataFrame`'s type. This is still a workaround rather than full support for typed `DataFrame`s. For complete support, external tools such as Strictly Typed Pandas [69] can be utilized.

Strictly Typed Pandas. *Strictly Typed Pandas* [69] introduces its own version of a `DataFrame` called `DataSet`, which is immutable and works with mypy. Casting to regular `DataFrames` is possible using the `.to_dataframe()` method. The package allows assigning types to `DataSet` columns, as demonstrated in the following code snippet:

```
from strictly_typed_pandas import DataSet
import numpy as np

class SchemaA:
    name: str

class SchemaB:
    id: int
    name: str

def create_dataset(names: DataSet[SchemaA]) -> DataSet[SchemaB]:
    n = len(names)
    ids = np.array(range(n), dtype=int)
    return DataSet[SchemaB](names.assign(id=ids))

df = DataSet[SchemaA]({"name": ["John", "Jane", "Jack"]})
ds = create_dataset(df)
print(ds)
print(ds.dtypes)

# Python Console
#    name  id
# 0  John   0
# 1  Jane   1
# 2  Jack   2
# name    object
# id       int32
# dtype: object
```

In the code snippet above, the first import of the necessary modules is necessary. Below, two classes carrying type information – `SchemaA` and `SchemaB` are defined.

The `create_dataset` function takes a `DataSet` that is structured according to `SchemaA` and returns a `DataSet[SchemaB]`. Inside the function, based on the length of the input `DataSet`, a NumPy array of integers with that length is created. On return, a new `DataSet[SchemaB]` is created with the array of integers serving as identifiers. Below, the function in action can be seen with the resulting printout at the very bottom.

While this approach does not support 100% of Pandas types, it does provide support for a common type with heterogeneous columns and homogeneous rows. This might be a good compromise between expressive power and concise syntax. Additionally, the ability to check declarations, function calls, and their return values with mypy should not be overlooked.

Overall, this example demonstrates how type annotations fulfill their original purpose of documenting code, as declared in Subsection 3.1.2. Moreover, this is achieved at a relatively small cost in terms of brevity.

3.1.6 Tooling

While documentation is an important aspect of type annotations, another significant benefit is error detection and prevention (see Subsection 3.1.2). To fully realize this benefit, a type checker is required. As discussed throughout this section, Python's authors did not intend to include a type checker in the language's core. Instead, their declared goal was to support external type

checkers. This section will review several prominent type checkers available for Python today.

Typeshed. `Typeshed` serves as a foundation for type checkers, as it is a central repository for static type definitions for Python. It implements PEP561 (described in Subsection 3.1.4). This GitHub repository is maintained by a community of contributors, including members of the Python development team. It is used by all the major type checkers described below to obtain type annotations for the language core, standard library, and widely-used external libraries. By providing a central source of type annotations, Typeshed enables the type checkers to use the same type definitions and keep up with language development [70].

Anyone is welcome to contribute to Typeshed (at the moment of this writing, 1169 contributors have done so). For adding a stub for third-party packages, the requirements are that the packages must be publicly available on the PyPI, support any Python version supported by Typeshed, and not ship with their own stub or type annotations. For larger changes, it is recommended to start by opening an issue outlining the proposed changes to get community feedback before investing significant time. Once the new change is ready for submission, the usual GitHub pull-request flow is used. That is, anyone interested may review the new code. One of the maintainers will merge the pull request when they think it is ready [71].

Typeshed annotates various constructs such as variables and constants, functions, classes, and type aliases. The annotations are stored in the form of stubs, which are non-runnable code saved as `.pyi` files. These stubs provide type information without including the actual runnable code [70].

For instance, consider the following examples of type annotations from Typeshed [70]:

- Variables and constants:

```
e: float
```

- Type aliases:

```
_SupportsFloatOrIndex: TypeAlias = SupportsFloat | SupportsIndex
```

  Where `SupportsFloat` is a protocol that represents types that can be converted to the `float` using the built-in function `float()`, while `SupportsIndex` is a protocol that represents types that can be converted to the `int` using the built-in function `index()`.

- Functions:

```
def acos(__x: _SupportsFloatOrIndex) -> float: ...
```

- Classes:

```
class _IsoCalendarDate(NamedTuple):
    year: int
    week: int
    weekday: int
```

  Where `NamedTuple` is a special base class for creating tuple classes with named attributes and type annotations.

By providing type annotations for various constructs, Typeshed allows external type checkers to understand the expected types and catch potential type-related issues in code.

```
number = input("What is your favourite number?")
print("It is", number + 1)
# error: Unsupported operand types for + ("str" and "int")
```

Code listing 3 Demonstration of the mypy type checker [73]

```
def f():
    return "PyCon"
def g():
    return f() + 2019

# pytype: line 4, in g: unsupported operand type(s) for +: 'str'
# and 'int' [unsupported-operands]
```

```
from typing import List
def get_list() -> List[str]:
    lst = ["PyCon"]
    lst.append(2019)
    return [str(x) for x in lst]

# mypy: line 4: error: Argument 1 to "append" of "list" has
# incompatible type "int"; expected "str"
```

Code listing 4 Demonstration of the PyType type checker [75]

mypy. Mypy (demonstrated in Listing 3), an open-source project developed by Jukka Lehtosalo and the Python community, was first introduced in 2012 [72]. It is a static type checker that focuses on compile time type checking and supports gradual typing, checking only annotated code. Mypy is strict and does not allow operations that change types. It can be considered a referential type checker for Python as many influential PEPs directly reference it (for instance PEP 484 [50], PEP 526 [53], and PEP 589 [58]).

PyType. PyType, an open-source project developed by Google, was first published on PyPI in 2016 [74, 75]. It is a static analyzer that does not execute the code. It attempts to check all code, including unannotated code, using type inference. PyType is more lenient than mypy and allows type mutations.

Listing 4 showcases the difference in the approaches taken by mypy and PyType. In the first example, PyType detects an unsupported operand for the addition operation that would be missed by mypy. In the second example, mypy detects an incompatible type when appending an integer to a list of strings, *pytype* would run the code with no issue displaying its leniency.

PyRight. PyRight, developed by Microsoft, is a type checker designed with performance in mind. It is often 3 to 5 times faster than mypy when type-checking large codebases. PyRight features a lazy, just-in-time type evaluator, recursive type evaluation, and its own parser that is able to recover from syntax errors and continue parsing. It was first published on PyPI in 2021 [76].

PyRight checks both annotated and unannotated code [77]. Within type inference, PyRight utilizes union types to ensure that the inferred type is neither too narrow nor too wide. Listing 5

```python
class A:
    def method1(self) -> None:
        self.x = 1

    def method2(self) -> None:
        self.x = ""
        # Mypy treats this as an error
        # because `x` is implicitly declared as `int`

a = A()
reveal_type(a.x)  # pyright: int | str
```

```python
def func1(val: object):
    if isinstance(val, str):
        pass
    elif isinstance(val, int):
        pass
    else:
        return
reveal_type(val)  # mypy: object, pyright: str | int
```

Code listing 5 Demonstration of the PyRight type checker [77]

demonstrates this behavior. In the first example, PyRight correctly infers the type of `a.x` as a union of `int` and `str`. Mypy would treat the assignment of an empty string to `x` as an error because `x` is implicitly declared as `int`. In the second example, PyRight narrows down the type of `val` to a union of `str` and `int`. Mypy would keep the type as `object` which is unnecessarily wide.

Pyre. Pyre is an open-source type checker developed by Meta (formerly Facebook) [78]. It is designed to be performant on large codebases with millions of lines of Python. The authors claim that Pyre is fast, integrated, fully featured, and built for security.

In addition to type checking, Pyre ships with Pysa, a static analysis tool. Pyre is capable of performing static type inference, scanning code, and automatically applying annotations [79]. The first version of Pyre was published on PyPI in 2018 [80].

The emphasis on security is demonstrated by the inclusion of Strict Mode. This configuration option allows enforcing type annotations either on the module or project level. Listing 6 illustrate the difference between non-strict and strict code.

In summary, the tooling ecosystem for Python's gradual typing is diverse and well-developed. Typeshed serves as a foundation for type checkers, providing a central repository for type annotations. Various type checkers such as mypy, PyType, PyRight, and Pyre cater to different needs and preferences, offering a range of features, strictness levels, and performance characteristics. This variety of tools highlights the commitment of the Python community to gradual typing and ensures that developers have ample options for type-checking and error detection. As the language and its typing system continue to evolve, the tooling ecosystem will likely adapt and expand, further enhancing the benefits of gradual typing in Python.

```
from typing import List

def unannotated():          # implictly returns `Any`
    return b"" + ""          # function body is not checked

def annotated() -> List:    # explicit return annotation - `annotated` is checked
    any = unannotated()
    any.attribute           # `Any` has all possible attributes
    return 1                # Err: returning `int` but expecting `List`
```

```
# pyre-strict
from typing import List

def unannotated():          # Err: missing return annotation
    return b"" + ""          # Err: function body *is* checked

def annotated() -> List:    # Err: implicit `Any` for a generic parameter to `List`
    any = unannotated()
    any.attribute           # Note: the type of `any` is still any.
    return 1                # Err: returning `int` but expecting `List`
```

Code listing 6 Demonstration of the Pyre type checker [79]

3.1.7 Evaluating Python's (mypy) Type System

Mypy as a reference type checker will be used for discussions and comparisons in the following text, including this discussion on type safety and type soundness.

Type Soundness. However, Ingkarat et al. [81] highlight that Python's type system (together with mypy or pytype) is unsound due to missing runtime checks and the fact that subtype checks involving the `Any` type always succeed.

Type-Enforcement Strategy. Weissmann et al. [24] categorize mypy's type enforcement strategy as "erasure", meaning that types are used for static analysis only.

In summary, Python's type system, when used with mypy, offers dynamic type safety while being type unsound. This is a consequence of the chosen type-enforcement strategy.

3.1.8 Summary

The design of Python feels cohesive, following clear guiding principles. However, even after 19 years, the work on implementing gradual typing is far from done, highlighting the complexity of such an undertaking.

The gradual typing process for Python began with discussions involving the community. The authors consistently reassured the community that it would still be possible to write untyped code, and the changes would not be imposed on them, even by convention. After a series of blog posts, discussion forum threads, and email exchanges, a formal proposal for enhancing the language was created.

Initially, the syntax for annotating functions was added. Only after that was the typing module introduced to define the semantics. Subsequently, annotations for variable declarations were introduced. Recent improvements focused on optimizing performance, notation, annotating complex data types, and resolving edge cases. Type-checking is still not part of the language.

Instead, this functionality is supplemented by various external packages offering an array of features and benefits, allowing programmers to select the one that suits them best or to not use any at all.

Key learnings from this process include the importance of engaging the community in discussions, responding to their feedback, and gradually enhancing the proposal. It is acceptable to initially lack type checking and other features. It is also reasonable to start with a less performant version and optimize it later. The key is to learn, adapt, and refine the system over time.

3.2 Ruby

Ruby is an open-source language originally developed by Yukihiro "Matz" Matsumoto in 1995 [82]. It is a dynamic, pure object-oriented language with complex but expressive grammar, influenced by Lisp, Smalltalk, Perl, C, and Java [83, p. 2]. The primary goal of Ruby is to make programming faster and easier. In Ruby, every value is an object, including simple numeric literals, true, false, and nil. The language enforces strict encapsulation, meaning there is no access to the internal state of an object from outside the object. A class in Ruby is a collection of methods that operate on the state of an object, with the object's state held by its instance variables [83, p. 2, 3, 8].

One of Ruby's distinguishing features is its support for metaprogramming. Metaprogramming allows developers to write code that generates, manipulates, or modifies other code, often at runtime. This can lead to more concise, flexible, and reusable code. However, metaprogramming presents a challenge for static type checking, as the code can change during runtime, making it difficult to analyze its behavior and properties. Given that a method can be added to a class during runtime, it is non-trivial to know whether the class does or does not have the method at the call site during the compile time [84].

Ruby was first made public with the release of ruby-0.95 on 21 December 1995 [85], and Ruby 1.0 was released on 25 December 1996 [85]. Initially, the language gained popularity in Japan, and its development was further advanced with the release of Ruby 2 in 2013 [86] and Ruby 3 in 2020 [87]. The Ruby on Rails web framework, developed by David Heinemeier Hansson of the 37signals company, has significantly contributed to Ruby's popularity [88]. Rails is a web-based, opinionated, full-stack framework for both front-end and back-end development.

Similar to Python, discussions about the language's further development take place on mailing lists [82]. In 2006, Ruby achieved mass acceptance, with active user groups formed in the world's major cities and Ruby-related conferences filled to capacity [82]. As of April 2023, Ruby is ranked 18th on the TIOBE Programming Community index [11], 17th in Stack Overflow's 2022 Developer Survey in the programming, scripting, and markup languages category [10], and 10th on GitHub's list of most-used programming languages in 2022 [9]. Ruby continues to be a popular choice for developers due to its flexibility, expressiveness, and strong community support.

3.2.1 Ruby Built-in Types

Ruby is a pure object-oriented language. All values are objects and all objects inherit from a class named Object and share the methods defined by that class [83, p. 86].

```ruby
class A
  attr_accessor :attr_a

  def initialize
    @attr_a = 'Hello'
  end
end

a = A.new

def a.greet
  puts "#{attr_a} World!"
end

a.greet  # 'Hello World!'

class B
  def greet
    puts 'Gutten Tag!'
  end
end

def say_hi(x)
  x.greet
end

b = B.new

say_hi(a) # 'Hello World!'
say_hi(b) # 'Gutten Tag!'

class DynamicGreeter
  attr_accessor :greeting

  def initialize(greeting)
    @greeting = greeting
  end

  define_method :greet do
    puts "#{greeting} Le Monde!"
  end
end

dynamic_greeter = DynamicGreeter.new('Bonjour')
dynamic_greeter.greet  # 'Bonjour Le Monde!'
```

Code listing 7 Demonstration of Ruby object-oriented system

Numeric Types. Numeric types in Ruby inherit from the `Numeric` class. The numeric classes in Ruby include `Integer` [89], representing integer numbers. Although other integer types such as `Fixnum` and `Bignum`, are no longer supported, they can still be used in the text below to demonstrate legacy tools in a time-appropriate manner. The `Float` [90] class represents real numbers while the `Complex` [91] and `Rational` [92] classes represent complex and rational numbers, respectively. The Ruby documentation and accompanying code snippets demonstrate a type hierarchy in which `Integer` is a "narrower" type than `Rational`, which is a "narrower" type than `Float`, which in turn is a "narrower" type than `Complex`. This hierarchy is similar to that found in Python.

Enumeration Types. Ruby does not have any enumeration types. Boolean values are not part of a separate enumeration type but are singleton instances of two separate classes: `true` as a singleton instance of the `TrueClass`, and `false` as a singleton instance of the `FalseClass` [83, p. 72].

Subrange Types. Subrange types are represented by a range object (1..10, 1...10). This Ruby syntax represents values between two numbers and is an instance of the `Range` class [83, p. 68].

Composite Types. Composite types in Ruby encompass a variety of data structures, including `String` [83, p. 46], a mutable sequence of characters; `Array` [83, p. 64], a mutable, indexed, heterogeneous, dynamically resizable sequence; and `Hash` [83, p. 67], an associative array of key-value pairs. Implicit conversions between types and other objects are possible as long as they have the appropriate method (`to_ary`, `to_int`, `to_s`, etc.).

The Listing 7 demonstrates some key features of its object-oriented system. Lines 1–7 define a simple class `A` with an attribute `attr_a` and an initializer method. Line 9 demonstrates how to instantiate a class. Lines 11–15 demonstrate Ruby's support for dynamically adding methods to objects. Lines 17–30 showcase Ruby's approach to duck-typing. Finally, the example demonstrates Ruby's metaprogramming capabilities with the `DynamicGreeter` class (lines 32–42). This class has a custom `greeting` attribute, and its `greet` method is defined dynamically during class definition using the `define_method` method (lines 39–41). On lines 44 and 45 a new instance of the `DynamicGreeter` is created with the greeting "Bonjour", and its `greet` method is called, printing "Bonjour Le Monde!".

In summary, Ruby's built-in types cover numeric, enumeration, subrange, and composite types, each providing unique features to the language. The object-oriented system, dynamic method addition, duck-typing, and metaprogramming capabilities demonstrated in the listing showcase Ruby's flexibility and expressiveness. This foundation sets the stage for discussing gradual typing, which builds on these core concepts to add type annotations and enhance the language's safety and reliability.

3.2.2 Towards Gradual Typing

The following text will delve into the history of gradual typing in Ruby, discussing the progression of type annotations and gradual typing systems within the language. The journey of gradual typing in Ruby has diverged significantly from Python's, with the Ruby core team making distinct design decisions that reflect an alternative philosophy and approach. This presents a unique opportunity to analyze how these differing perspectives, compared to Python, have shaped the gradual typing landscape in Ruby.

Figure 3.2 illustrates the history of Gradual Type Checkers for Ruby, discussed further in this text. Individual tools are mapped to their year of creation, either through the publication of a paper or the release of a production version of the tool on *RubyGems* – a package management system for Ruby. The figure also presents major versions of Ruby and adjacent academic work.

An arrow on the diagram indicates that the author directly referred to the previous work in an adjacent paper or that part of the previous work (typically type definitions) was adapted in the later one.

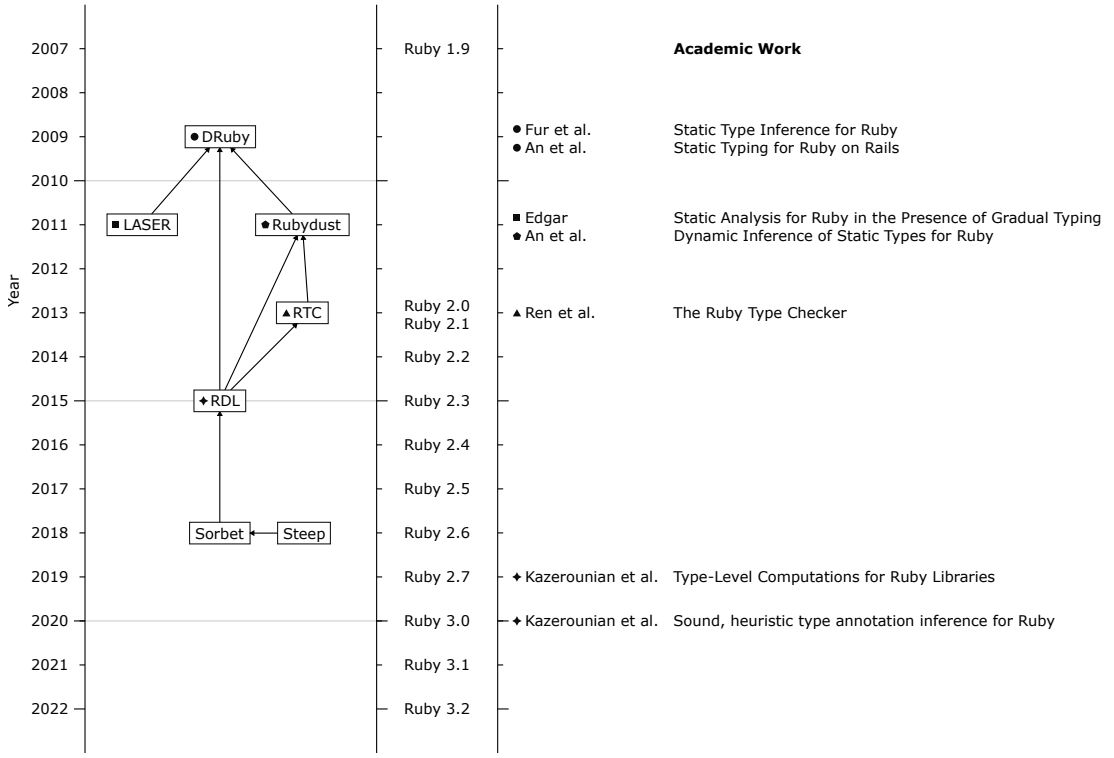

Figure 3.2 History of Gradual Type Checkers and Adjacent Academic Work

Furr et al. [93] noted in their 2009 paper that Ruby had been known for its steep learning curve and focus on the development experience. The price for this had been that errors might have remained latent until runtime. They observed that introducing type annotations could have improved code documentation and made it easier to identify type-related errors before they became runtime issues.

3.2.2.0.1 Diamondback Ruby To address this issue, they introduced an extension for Ruby called *Diamondback Ruby* (*DRuby*). This extension added a new static type inference system to enhance Ruby's type system. The goal had been to improve code safety, maintainability, and readability by allowing developers to provide type annotations [93].

The type annotations were in the form of comments to ensure interoperability with the standard Ruby interpreter. The DRuby annotation language offered features such as intersection types, union types, optional types, self type, parametric polymorphism or generics, and tuple types for heterogeneous arrays [93].

In the example below, the `+` operator and the `insert` method are annotated with comments indicating their expected argument types and return types [93]:

```
class String
...
##% "+" : (String) -> String
def +(p0); end
##% insert : (Fixnum, String) -> String
def insert(p0, p1); end
...
end
```

This example demonstrates intersection types. The `include?` method allows both `Fixnum` and `String` types, working similarly to overloaded functions in other languages. The method's behavior differs depending on the argument type [93]:

```
...
##% include? : (Fixnum) -> Boolean
##% include? : (String) -> Boolean
def include?(p0); end
...
```

This one demonstrates union types. The `%` method takes any of the three numeric types (`Fixnum`, `Float`, or `String`). This feature serves cases where multiple types share the same method(s) called within the function, making them substitutable for the method's purpose [93]:

```
...
##% "%" : (Fixnum or Float or String) -> String
def %(p0); end
...
```

The following two examples are functionally equivalent and demonstrate optional types [93]:

```
##% chomp : () -> String
##% chomp : (String) -> String
def chomp(p0=$/); end
```

```
##% chomp : (?String) → String
def chomp(p0=$/); end
```

This example demonstrates the use of the `self` type as a reference to the adjacent object [93]:

```
##% clone: () -> self
def clone() end
```

The following example shows parametric polymorphism. In this example, the array is of a generic type, which means the return value of the method `at` is the same type as the `<t>` type of the array [93]:

```
##% Array<t>
class Array
  ##% at : (Fixnum) -> t
  def at(p0); end
  ...
end
```

Tuple types are used to handle heterogeneous arrays, where each item can be of a different type. Essentially, tuple types are vectors of types that map 1:1 to the items in the array.

```
##% f : () -> (Fixnum, Boolean)
def f(); [1, true] end
a, b = f
```

After its introduction, DRuby was extended to support the Ruby on Rails framework [94]. After that, however, there was not much news. The last update on the research project's official web page is from 2009 [95]. Even though it seems like DRuby never reached wide adoption, it proved that static typing in Ruby is possible.

LASER: Lexically and Semantically Enriched Ruby. LASER, a thesis project by Michael Joseph Edgar, aimed to determine what useful information about real-world Ruby programs could be statically discovered. LASER leveraged traditional static analysis techniques, such as Flow Analysis [96] and Static Single Assignment [97] transformations, to extract meaningful program invariants, including both explicitly programmed constants and those implicitly defined by Ruby's semantics [84]. By establishing a well-defined distinction between load time and runtime, LASER could determine the properties of a Ruby program even when it included common metaprogramming techniques [84].

Inspired by DRuby and Ripper, a Ruby feature capable of printing out an AST, LASER performed static analysis and linting for Ruby code [84]. It utilized Ripper to create an AST and conducted both low-level and high-level analysis, including Control Flow Graph (CFG) generation, Top-Level Simulation, Static Single Assignment, Type Inference, Purity Detection, Constant Propagation, Inferring Block Use Patterns, Unreachable Code Analysis, and Unused Variable Analysis [84]. Finally, LASER generated a printout of warnings and errors based on its analysis. Although the author mentions type annotations, it appears that they were not included in the final version of the project [84].

Below is an example of LASER's analysis of a Ruby code snippet [98]:

```
1   $ cat temp.rb
2   class Foo
3     def initialize(x, *args)
4       a, b = args[1..2]
5     end
6   end
7   Foo.new(gets, gets)
8
9   $ laser temp.rb
10  4 warnings found. 0 are fixable.
11  ================================
12  (stdin):3 Error (4) - Variable defined but not used: x
13  (stdin):3 Error (6) - LHS never assigned - defaults to nil
14  (stdin):3 Error (4) - Variable defined but not used: a
15  (stdin):3 Error (4) - Variable defined but not used: b
```

Lines 2–7 show the Ruby code being analyzed. Line 10 provides high-level statistics generated by LASER. Lines 11–15 elaborate on individual errors. Line 11 points out that variable `x` is defined but never used, while line 12 indicates that the Left-Hand Side (LHS) of the assignment is not assigned a value, and so it defaults to 'nil'. Lines 13 and 14 reveal that variables `a` and `b` are defined but not used in the code.

Although there was no development after 2011, the project has 390 stars and 16 forks on GitHub.

Rubydust. *Rubydust*, introduced in 2011, was a dynamic analysis tool for Ruby aimed at providing static guarantees, such as eliminating method-not-found errors [99]. Among the authors are Jong-hoon An, Jefferey S. Foster, and Michael Hicks who previously worked on DRuby. Rubydust overcame the challenges of static analysis in Ruby by utilizing extensive test suites, method call boundaries, and constraint-based dynamic type inference. This allowed for inferring correct types and detecting type errors while maintaining soundness qualified by coverage.

In summary, Rubydust offered a novel approach to dynamic type inference for Ruby, implemented as a metacircular library, and yielded promising results on small programs [99]. The work on Rubydust subsequently inspired the development of the Ruby Type Checker (*RTC*) [100].

The Ruby Type Checker. RTC was introduced in 2013 by a team of researchers, including Jeffrey S. Foster [100]. Inspired by the dynamic type inference tool Rubydust [99], RTC focused solely on runtime analysis, which might technically disqualify it as an example of gradual typing. However, considering that Ren et al. [99] were adding type annotations to a dynamically typed language, running type checks earlier (though still during runtime), and allowing both annotated and unannotated code to coexist within the same file, the author finds RTC relevant to the topic of this thesis.

RTC supported annotations on classes, methods, and objects. Its type system included union types, intersection types, higher-order method types, polymorphism, and type casts. The implementation used annotated objects wrapped by proxy objects that connected types with the underlying object. When a proxy was instantiated, it executed the checks before and after delegating the call to the underlying object. This meant that the type-checking happened eagerly, i.e., when the method was called. The proxy also annotated incoming arguments and the return value [100].

Annotations occur on a class basis. A call to `rtc_annotated` needs to be made to mark a class as being annotated:

```
class A
    rtc_annotated
end
```

This call makes annotation methods available locally. One such annotation method is `typesig`:

```
typesig "method_A: () -> Fixnum"
def method_A ... end
```

This annotation specifies `method_A` as having no parameters and a return value of `Fixnum` type. Union types are annotated using the keyword `or`:

```
typesig "method_B: () -> Fixnum or %false"
def method_B() ... end
```

Where `method_B` returns either `Fixnum` or `false`. Intersection types are annotated using multiple method annotations one above another:

```
typesig method_C: (A) -> Fixnum
typesig method_C: (Fixnum) -> Fixnum
def method_C(employee) ... end
```

Where `method_C` accepts either `A` (an employee object) or `Fixnum` (an employee ID). Higher-order methods can be annotated using the following syntax:

```
class String
    rtc_annotated
    typesig "each_char: () { (String) -> %any } -> String"
end
```

In this example, the `String` class defines the method `each_char`, which calls its higher-order argument on each character of the receiver as a string of length 1. Since the return value of the higher-order method is not used by `each_char`, `%any` is used to signify that it might return any value. Parametric polymorphism can be achieved using a type variable:

```
class Array
    rtc_annotated [:t, :each]

    typesig " '[]': (Range) -> Array<t>"
    typesig " '[]': (Fixnum, Fixnum) -> Array<t>"
    typesig " '[]': (Fixnum) -> t"
end
```

Where the `:t` is a type parameter. `:each` indicates how to find the contents type of `:t` for a raw `Array` – when checking whether a raw `Array` can be annotated with type `Array<u>`, the `each` method is called to iterate over all elements and test their type. Classes with multiple type parameters can be specified by passing multiple two-element list arguments to `rtc_annotated`. This was not very fast – that is why RTC allows opting out from this type of checks by using a non-strict mode.

The performance overhead of RTC was large, often several orders of magnitude. The non-strict mode showed a speed-up, especially when the program performed many Array operations. The most significant recorded speedup was about 47%, still far from the speed of unannotated code. However, the authors were able to discover some type errors [100].

After its introduction, RTC gained some adoption, with 84 stars and 4 forks on GitHub. However, there was not much development since then. The only new code pushed to GitHub was a 2016 announcement that RTC had been superseded by a new tool called *RDL* [101].

RDL. RDL was a lightweight system for adding types, type checking, and contracts to Ruby programs [102]. First introduced in 2015 by the Programming Languages Research group at the University of Maryland. Among the authors are Jeffrey S. Foster together with Brianna M. Ren and Stephen T. Strickland who previously coauthored RTC. RDL authors also referred to DRuby and Rubydust as their sources of inspiration [103].

Although the term ”gradual typing” was not explicitly mentioned, RDL allowed for the coexistence of statically and dynamically typed code, as well as a smooth transition between the two. By meeting these criteria, RDL aligns well with the definition established in Section2.2. Considering the focus of this thesis, all non-typing features of RDL will be ignored in the text below.

RDL provided type definitions for Ruby core and Ruby on Rails. The syntax of RDL was similar to RTC, with annotations added above method definitions. Type checking could be performed either at runtime or statically [103]. RDL supported union types, intersection types, higher-order methods, and parametric polymorphism.

RDL's syntax included a variety of features for adding type annotations to Ruby code [103]. To add annotation methods to the current scope, the following code is used:

```
require 'rdl'
extend RDL::Annotate
```

RDL, similar to RTC, decorates methods by adding annotations above them:

```
type '(Integer, Integer) -> String'
def method_A(x, y) ... end
```

This gets checked at runtime. It is also possible to check statically:

```
type '(Integer) -> Integer', typecheck: :label
def method_B(x)
  "forty-two"
end

RDL.do_typecheck :label
```

An error message is generated if a type mismatch is detected:

```
$ ruby file.rb
.../lib/rdl/typecheck.rb:158:in `error':  (RDL::Typecheck::StaticTypeError)
.../file.rb:6:3: error: got type `String' where return type `Integer' expected
.../file.rb:6:   "forty-two"
.../file.rb:6:   ^~~~~~~~~~~
```

Union types are annotated as follows:

```
type '(Numeric or String) -> %any'
def method_C(x) ... end
```

Here, `x` can be either `Numeric` or `String`, and the return type `%any` is a "top" type that matches any object. Intersection types are annotated like this:

```
type '(A) -> Integer'
type '(Integer) -> Integer'
def method_D(employee) ... end
```

Method `D` accepts either `A` or `Integer` (we can imagine `A` class representing an employee or an ID of such an employee). Higher-order methods are annotated like this:

```
type '() { (String) -> %any } -> String'
def each_char() ... end
```

In this example, method `each_char` calls its higher-order argument on each character of the receiver as a string of length 1. Since the return value of the higher-order method is not used by `each_char`, `%any` is used to signify that it might return any value. Parametric polymorphism is handled like this:

```
class Array
  type_params [:t], :all?

  type Array, :[], '(Range) -> Array<t>'
  type Array, :[], '(Integer, Integer) -> Array<t>'
  type Array, :[], '(Integer or Float) -> t'
end
```

The `type_params` method names the type parameter of the class. The three type annotations show three acceptable signatures for method `[]`. To assign a proper type to an `Array` object, the programmer needs to use the `RDL.instantiate!` method:

```
x = [1, 2]
RDL.instantiate!(x, 'Integer')
x.push('three') # type error
```

`Type_params` support variance, where `type_param [:t], :all? :+` would be covariant, `type_param [:t], :all? :-` contravariant, and `type_param [:t], :all? :~` invariant [103].

In 2019, Kazerounian et al. introduced CompRDL, an extension of RDL that allowed library method type signatures to include type-level computations, singleton types for table and column names, and precise type signatures for tables, hash, array, and strings [104]. CompRDL was proven to be sound and allowed for type-checking database queries.

RDL's ideas have influenced the industry-standard type checker Sorbet which even adapted RDL's standard library definitions [105]. It also influenced Ruby 3 [106]. With over 35,000 downloads and 602 stars and 37 forks on GitHub, RDL has become a popular tool for adding gradual typing features to Ruby programs [102, 103].

3.2.3 Emergence of Sorbet

Table 3.1 Early Development of Sorbet at Stripe

October 2017	Project kickoff, existing tools analysis, prototyping
November 2017	Go/no-go date, type syntax and type system features decided
February 2018	First code annotated manually
April 2018	60-day "dark launch", type checking added as a non-blocking step in CI
May 2018	Official announcement on Ruby Kaigi 2018
June 2018	Enforced in CI for every Stripe engineer
January 2019	Support for editor and OSS (Open Source Software) tooling
June 2019	Sorbet open sourced

In 2016, a full year before the work on Sorbet began, the first type annotations emerged in Stripe's (a payment company) Ruby codebase, which were checked at runtime with a check being performed on each invocation [107]. The code looked like this:

```
class Yell < Opus::Command
    declare_method({msg: String}, returns: String)
    def call(msg)
        helper(msg)
    end

    def helper(msg)
        "Yell: #{msg}"
    end
end
```

As Stripe's Ruby codebase grew rapidly, the technological decisions that had helped them iterate faster in their early days started to slow them down. By 2017, onboarding new engineers took longer, and implementing new changes became difficult, some even borderline impossible. The team decided to stick with Ruby, but some changes needed to be made. They seriously considered adopting one of the two existing type checkers, RDL or TypedRuby, but neither met their needs. RDL, while powerful, was just too slow for the needs of the payment company. TypedRuby was faster but buggy and offered only limited functionality. Inspired by Microsoft's

TypeScript and Facebook's Hack, they decided to try a similar approach with Ruby and started the development of Sorbet [107].

In 2018, Petrashko et al. presented Sorbet to the public for the first time at Ruby Kaigi, a conference focused on the Ruby ecosystem. They described the situation at Stripe and the scale of Ruby usage in the company, with millions of lines of code. They also launched a public demo and announced plans to make Sorbet open-source. Sorbet's design principles were as follows[108]:

- *Explicit* – that means annotations
- *Useful, not burdensome* – the syntax should be concise, error messages clear
- *As simple as possible, but powerful enough* – the type system should be expressive enough to cover a big portion of real Ruby code. On the other hand, the goal is not to type all constructs at any cost. The type system needs to be simple, easy to learn, and easy to understand.
- *Compatible with Ruby* – no new syntax
- *Scales* – should enable speed for teams and codebases of any size
- *Can be adopted gradually*

Even back then, Sorbet offered an impressive list of features such as the identification of incompatible types, non-existent methods, and unreachable code. It provided three levels of strictness[108]:

- `# typed: true` - enabled type checking
- `# typed: strict` - required instance variables to be declared
- `# typed: strong` - disallowed calling untyped code

Sorbet enabled annotating methods[108]:

```
sig(param1: T1, param2: T2).returns(T3)
def method_X(param1, param2) ... end
```

Where `method_X` accepts two parameters of types T1, and T2 respectively. Annotating variable declarations was optional[108]:

```
a = 5
a = T.let("str", String)  # Explicitly declared type String
```

Where the first line infers type `Integer`, while the second performs explicit re-declaration to the `String` type. Generic classes could be annotated as follows[108]:

```
class Box
  extend T::Generic

  Elem = type_member

  sig.returns(Elem)
  attr_reader :x

  sig(x: Elem).returns(Elem)
  attr_writer :x
end

int_box = Box[Integer].new
```

Where `Elem` is the generic type. Generic methods would be annotated as follows[108]:

```
class Array
  Elem = type_member

  type_parameters(:U).sig(
      blk: T.proc(arg0: Elem).returns(T.type_parameter(:U)),
  )
  .returns(T::Array[T.type_parameter(:U)])
  def map(&blk); end
end
```

In this example, `Elem` is the generic type representing the elements of the array, and the `map` method is defined as a generic method that takes a block (higher order method) `blk` with a single argument of type `Elem`. The block returns a type `U`, and the `map` method itself returns an array of type `U`. This allows the type checker to infer the type of the result of the `map` method based on the block passed to it.

In the process of developing Sorbet, the team made several key decisions, such as changing the syntax from

```
sig(bar: Integer).returns(String)
```

to

```
sig {params(bar: Integer).returns String}
```

to allow for lazy loading of type definitions. They also introduced new tools for translating dynamic definitions to static ones, automatically fixing common bugs, identifying type-checkable files, and determining the most impactful methods to type [105].

By May 2019, all of Stripe's Ruby developers used Sorbet to check new code, with a large portion of older code being annotated as well [109]. A pilot program was conducted with crypto exchange Coinbase and e-commerce platform Shopify [109]. Sorbet was introduced to rubygems.org in 2018 as version 0.0.1.pre.prealpha, with the first production version (0.0.42) being released in 2019 [110].

At the 2019 Kaigi conference, new IDE features were announced, such as "Go to Definition," which allows developers to go from an instance to the definition of an expression, autocomplete functionality, and contextual documentation display [111]. It was also announced that the Sorbet team was collaborating with the Ruby team on types for Ruby 3.0, and new documentation was released. Additional tooling was introduced, including the *srb rbi* (a family of commands for automatic generation of `.rbi` files), improved support for the Ruby on Rails framework, and *sorbet-typed*.

Sorbet-typed, developed by Coinbase, inspired by TypeScript's DefinitelyTyped repository, served as a central repository for Sorbet-compatible type definitions, stored as `.rbi` (Ruby Interface) files [112]. These type definitions were used by Sorbet for optional annotation of both Ruby core and external packages and were stored in a GitHub repository where Ruby package creators can submit their type definitions for use by the library.

RBI files provided information that Sorbet does not understand naturally, such as anything defined in a package, dynamically modified ancestors, constants accessed or defined with `const_get` or `const_set`, and methods defined with `define_method` or `method_missing` [113]. These type definitions could be generated or handwritten, and developers could update them as needed, giving them full control over their projects and ensuring predictability. Syntax of RBIs was the same as normal Ruby file, except for method definitions not needing implementation:

```
# -- example.rbi --
# typed: strict

module MyModule
end

# Declares a class
class A
  include MyModule
  extend MyModule

  X = T.let(T.unsafe(nil), Integer)

  sig {params(x: Integer).returns(String)}
  def method_A(x)
  end
end
```

Where `X` is an integer constant with arbitrary value and `method_A` is a method accepting `Integer`, returning `String`.

In 2019, the emergence of *Tapioca* was witnessed as an alternative to the use of *srb rbi*. Alongside this, *rbi-central* provided type definitions for Tapioca, presenting an alternative to *sorbet-typed*. Both of these tools were developed by the team at Shopify [114].

In a presentation at the 2019 JVM Language Summit, Petrashko shared benchmark results demonstrating Sorbet's performance, with an average incremental type-checking time of 5ms on keypress in the integrated development environment (IDE) [115]. He also discussed the implementation details.

The implementation details. Sorbet's type-checking process consisted of two main steps: syntactic analysis and semantic analysis. The syntactic analysis involved several phases, such as pre-processing and indexing of the source code (index phases), building an Abstract Syntax Tree (AST) from the source code (parser), simplifying the AST by rewriting complex constructs into simpler ones (desugar), handling common library constructs such as getters, setters, Ruby Structs, and others (DSL passes), and discovering definitions and resolving class names, ultimately building the SymbolTable (namer and resolver).

Semantic analysis was composed of two main phases: building a Control Flow Graph (CFG) and type-checking it, and inferring types for expressions and variables based on the information available from the preceding phases. The type-checking phase operated on an immutable global state, ensuring consistency. The type inference phase further refined the typing information and provided more precise type checking.

These implementation details contributed to Sorbet's impressive performance and allow it to effectively handle complex Ruby codebases while providing accurate type checking and error reporting.

Steep. Around the same time as Sorbet, work began on another gradual type checker called *Steep* [116]. Steep was developed by Soutaro Matsumoto (not to be confused with Yukihiro Matsumoto, the creator of Ruby). In the early days of Steep's development, limited information was available on the English-speaking internet. The first mention the author of this thesis was able to find was in a 2019 article on type checking in Ruby by IT journalist Michael Kohl [117]. Kohl noted that Steep was the most widely known alternative to Sorbet and, unlike Sorbet, Steep did not use annotations or type inference, relying entirely on `.rbi` files.

3.2.4 Ruby 3 and Beyond

RBS. Soutaro Matsumoto played a significant role in the development of Ruby 3, particularly in the area of type-related features [118]. He focused on creating a new language for type signatures called *RBS*. These type signatures were stored in `.rbs` files, separate from the Ruby code itself. The RBS library facilitated the portability of type signatures between type checkers and encouraged the community to write types for their packages and applications.

RBS type signatures had several key features, including duck typing, union types, and method overloading. These features allowed for increased code safety, better IDE integration, and the detection of more bugs. Both Steep and Sorbet announced support for RBS. Matsumoto, in the announcement of the new feature, made it clear that their aim was not to deprecate Sorbet or RBI but rather to achieve interoperability. RBS also included a translator for converting RBI files to RBS format [118].

The Sorbet team announced they would support both RBI and RBS files, allowing for individual preference [119]. They doubled down on their take on inline annotations reasoning: it prevented ambiguity, was fully compatible with other tools, and was backward compatible. However, there were some drawbacks, such as limiting options and the inability to extend Ruby's syntax for prettier annotations [119].

Ruby 3. Ruby 3 was released on Christmas day in 2020 with three main features: performance, concurrency, and typing [120]. The long-term vision of Matz (Yukihiro Matsumoto) was to achieve static type checking without type declaration, using abstract interpretation. Typing features included RBS, and TypeProf, an experimental type analysis tool using type inference to generate RBS files [120].

RBS was able to support basic constructs of the language [118]:

```
class Merchant
attr_reader token: String
attr_reader name: String
attr_reader employees: Array[Employee]

def initialize: (token: String, name: String) -> void

def each_employee: () { (Employee) -> void } -> void
| () -> Enumerator[Employee, void]
end
```

Where `Merchant` is a class with three attributes: `token`, `name`, and `employees`, that are typed `String`, `String`, and a generic class – `Array[Employee]` respectively. The class also has two methods – `initialize` that accepts two parameters of type `String` and returns `void` and `each_employee` that accepts a higher-level method (also called a block) and returns `void` or accepts `void` and returns an `Enumerator` instance. Duck typing support would be achieved, using interfaces [118]:

```
interface _Appendable
def <<: (String) -> void
end

def append: (_Appendable) -> String
```

Where `_Appendable` is an interface that requires `<<` operator that accepts a String object. Method `append` will then accept an `Array[String]` that has such an operator, but won't accept, for example, `Integer` or `True`. Part of the specification is union types with the following syntax [118]:

```
def author: () -> (User | Bot)
```

Where the `author` method returns either a `User` or a `Bot` type. The syntax would then go as follows [118]:

```
def each_reply: () -> Enumerator[Comment, void]
              | { (Comment) -> void } -> void
```

Where the `each_reply` method has two different signatures: one that accepts no arguments and returns an `Enumerator[Comment, void]`, and another that accepts a block taking a `Comment` object and returns void.

In 2021, the Sorbet team open-sourced an experimental ahead-of-time compiler [121]. Though not ready for external use, for Stripe's production API traffic it was 22–170% faster than Ruby's default implementation. The team chose ahead-of-time compilation because it is conceptually simpler, and allows static type checking to provide performance improvements.

Ruby 3.1.0, released in 2021, added more type-related features. RBS introduced bounded polymorphism and generic type aliases. The bounded polymorphism would work as follows [122]:

```
class PrettyPrint[T < _Output]
  interface _Output
    def <<: (String) -> void
  end

  attr_reader output: T
end
```

Where `T` is a type parameter that implements the `_Output` interface. The `_Output` interface require `<<` method accepting `String` and returning `void`. `String` provides such method while `Integer` does not. The generic alias would then look like this:

```
type list[T] = [T, list[T]] | nil

type int_list = list[Integer]      # List of Integer
type object_list = list[Object]    # List of Object
```

TypeProf in the same version added experimental IDE support, which included displaying guessed or provided method signatures above methods and completing method names [122].

In 2022, the Sorbet team open-sourced their extension for VS Code, a popular IDE [123]. That same year, the Sorbet team officially recommended Tapioca as the go-to package for generating RBI files, replacing the `srb rbi` family of commands. Sorbet-typed would then be superseded by rbi-central [124]. The `srb rbi` command entered maintenance mode, with no new features being added and the potential for retirement if a new Ruby release causes a breaking change.

3.2.5 Typed Ruby in 2023

Throughout this section, both academic research as well as commercial efforts on adding gradual typing to Ruby were discussed. Gradual type checkers such as Sorbet or Steep are popular and used by many programmers. At the moment, Sorbet seems to be ahead with almost 17 million downloads on rubygems [110], with Steep being a distant second with just over 800 thousand downloads [125].

Listing 8 demonstrates how the same code can be written in Ruby both with annotations using Sorbet and without annotations.

```ruby
# Untyped code


class Vec2D









  attr_accessor :x, :y




  def initialize(x, y)
    @x, @y = x, y
  end









  def +(other)
    if other.is_a?(Numeric)
      other = Vec2D.new(other, other)
    end

    Vec2D.new(
      self.x + other.x,
      self.y + other.y
    )
  end


  def to_s
    "(#{@x}, #{@y})"
  end
end

vec1 = Vec2D.new(1, 2)
vec2 = Vec2D.new(3, 4)
vec3 = Vec2D.new(1.5, 2.5)

puts vec1 + vec2  # Output: (4, 6)
puts vec1 + vec3  # Output: (2.5, 4.5)
puts vec1 + 2     # Output: (3, 4)
```

```ruby
# typed: strict
require 'sorbet-runtime'

class Vec2D
  extend T::Sig

  T_IntFloat = T.type_alias {
    T.any(Integer, Float)
  }

  attr_accessor :x, :y

  sig {
    params(x: T_IntFloat, y: T_IntFloat).void
  }
  def initialize(x, y)
    @x, @y = x, y
  end

  sig {
    params(other: T.any(Vec2D, T_IntFloat))
    .returns(Vec2D)
  }
  def +(other)
    if other.is_a?(Numeric)
      other = Vec2D.new(other, other)
    end

    Vec2D.new(
      self.x + other.x,
      self.y + other.y
    )
  end

  sig { returns(String) }
  def to_s
    "(#{@x}, #{@y})"
  end
end

vec1 = Vec2D.new(1, 2)
vec2 = Vec2D.new(3, 4)
vec3 = Vec2D.new(1.5, 2.5)

puts vec1 + vec2  # Output: (4, 6)
puts vec1 + vec3  # Output: (2.5, 4.5)
puts vec1 + 2     # Output: (3, 4)
```

Code listing 8 Comparison of Ruby code without and with type annotations

In the type-annotated version of the code (on the right), Sorbet type annotations enhance the code's readability for both humans and the type checker. The `# typed: strict` directive on the first line enforces strict typing rules, requiring instance variables to be declared. Sorbet's runtime system is imported on the second line, while `extend T::Sig` on the fifth line allows the `Vec2D` class to use the `sig` method for type annotations. A type alias, `T_IntFloat`, representing `Integer` and `Float` types, is defined on lines 7–9. The `initialize` method's type annotation on lines 13–15 specifies that it takes two `T_IntFloat` parameters and returns `void`. The operator `+` is annotated on lines 20–23, showcasing union types by accepting either a `Vec2D` or a `T_IntFloat` and returning a `Vec2D`. Lastly, the `+` method's body (lines 25–32) demonstrates different behaviors based on the parameter's type.

While this style of annotations is arguably neither the shortest nor easiest to read, it allows for full backward compatibility and interoperability with the existing Ruby ecosystem as well as excellent performance.

3.2.6 Tooling

Similarly to Python (see Subsection 3.1.6), type checking in Ruby is not integrated into the core language and relies on external type checkers and the adjacent ecosystem. In this subsection, we discuss several notable tools available to the Ruby community.

Sorbet. Sorbet is a massively popular gradual type checker, extensively discussed throughout this section. As of 2023, it is employed by dozens of companies, including Stripe, Shopify, Coinbase, Instacart, and Kickstarter [126]. Sorbet provides IDE support for VS Code and via *Language Server Protocol* to several others including Vim, Emacs, Sublime, or IntelliJ. With over 3.4k stars on GitHub and 16,995,789 downloads on RubyGems [110], Sorbet has become an essential part of the Ruby ecosystem.

Tapioca. Tapioca, a tool developed by Shopify [114], focuses on generating Ruby Interface (RBI) files for packages used in an application as well as various Domain Specific Language (DSL) patterns that rely on metaprogramming. Tapioca allows for the automatic generation of RBI files based on an application's dependencies file, importing signatures and documentation from the source code of packages, and synchronization validation for continuous integration. Tapioca also assists in managing shim RBI files and finding useless definitions in them. As of 2023, Tapioca has 518 stars on GitHub and 6,186,848 downloads on RubyGems [127].

Steep. Steep, another gradual type checker for Ruby developed by Soutaro Matsumoto [116], supports the RBS format for type annotations. Unlike Sorbet, Steep does not support inline type annotations; instead, annotations are stored in a separate file. Steep allows generating signature prototypes, providing a starting point for annotating files. An example of Steep's signature prototype generation is shown below:

```
$ rbs prototype rb lib/person.rb lib/email.rb lib/phone.rb
class Person
  @name: untyped
  @contacts: Array[untyped]
  def initialize: (name: untyped) -> Array[untyped]
  def guess_country: () -> untyped
end
```

With IDE support for VSCode and Sublime Text, Steep is actively developed and continues to gain traction. As of 2023, Steep has 1.2k stars on GitHub and 812,502 downloads on RubyGems [125].

These tools have significantly contributed to the adoption of gradual typing in the Ruby community and have become essential components of the Ruby ecosystem.

3.2.7 Evaluating Ruby's (Sorbet) Type System

Sorbet will be due to its popularity used for comparison and discussion in the rest of the text including this brief discussion on type soundness of its type system.

Type Soundness. Although the selected type-enforcement strategy allows for type soundness, there are documented cases where Sorbet behaves unsoundly [128].

Type-Enforcement Strategy. Greenman et al. [24] categorize Sorbet's type enforcement strategy as hybrid enforcement, as it performs static analysis while runtime checks are optional.

In summary, Sorbet's type system enforces type safety and aims for type soundness, though there are instances where it may behave unsoundly. Its hybrid type-enforcement strategy allows for both static analysis and optional runtime checks, catering to various development scenarios and requirements.

3.2.8 Summary

Ruby's dynamic type system is coherent and follows clear guiding principles. However, approaches towards the addition of gradual typing were quite different. The journey for Ruby began in the academic sphere, with nine years of experimental work preceding the emergence of the first commercial tool. These commercial tools are built heavily on previous academic work, providing a head start.

The academic tools excel in many aspects, such as the ability to annotate language constructs and (for some) soundness, but lack the runtime performance required for commercial applications like payment processing and e-commerce. In contrast to Python, the core Ruby language was slow to incorporate syntax changes supporting gradual typing, resulting in a less clean and more verbose syntax (compare Listing 2 to Listing 8). A significant challenge in adding gradual typing to Ruby was its extensive use of metaprogramming capabilities, which made type annotations more complex.

Despite these challenges, Ruby tools (especially Sorbet) have achieved impressive speed in the critical path of their domain-specific uses. Throughout the development process, the gradual typing journey has been driven more by companies than individual developers. The engagement between the language community and the Ruby core team played a crucial role in shaping the gradual typing system.

One unique aspect of Ruby's approach is the use of type annotations in separate files (RBI or RBS), with both major type checkers supporting this feature while only one of them supports annotations directly in code.

The experiences from Ruby's gradual typing journey offer valuable insights for adding gradual typing to R. Engaging the community and learning from feedback, is a key. Additionally, considering the balance between academic and commercial needs, and focusing on performance optimization for practical applications are other important factors. By adopting these lessons, R can benefit from a more robust and flexible type system, providing developers with improved code quality and maintainability.

3.3 PHP

PHP (PHP: Hypertext Preprocessor) is a popular programming language primarily used for web development, designed for creating HTML content and supporting object-oriented programming [129]. PHP allows for server-side scripting, command-line scripting, and GUI applications. The language is flexible and offers support for major databases such as MySQL, PostgreSQL, Oracle, and more. PHP has a rich standard library and a variety of extensions.

PHP was created by Rasmus Lerdorf in 1994, originally known as Personal Home Page [129]. The first version was announced in 1995 on the Usenet as a framework for useful tools. PHP 2.0, announced in 1996, was a server-side HTML embedded scripting language. This shift towards a scripting language was driven by PHP users who demanded greater control over their websites.

In 1997, Andi Gutmans and Zeev Suraski created PHP 3.0, which provided support for major operating systems, databases, and email protocols [130]. In 1999, Gutmans and Suraski established Zend Technologies and developed PHP 4.0 along with an open-source parser called Zend Engine. The company played a pivotal role in the development and popularization of PHP, contributing to its technical advancements and widespread adoption. In 2015, Zend Technologies was acquired by Rogue Wave Software, which was later acquired by Perforce Software in 2019, becoming Zend by Perforce [130].

PHP 5.0 was released in 2004, followed by PHP 7.0 in 2015 after PHP 6.0 was never released due to development issues [131]. PHP 8.0 was released in 2020. In 2021, the non-profit PHP Foundation was established to support, advance, and develop the PHP language [132].

Today, PHP is a popular and widely used programming language. It ranks 7th on GitHub's list of most-used programming languages in 2022 [9], 10th in Stack Overflow's 2022 Developer Survey in the programming, scripting, and markup languages category [10], and 9th on the TIOBE Programming Community index for April 2023 [11].

3.3.1 PHP built-in types

Like Python and Ruby, PHP is a dynamically typed language that includes several built-in types [133].

The type system employs a nominal approach with a strong behavioral subtyping relation, ensuring that a subtype's behavior is a more specific version of its supertype's behavior. This guarantee is achieved through compile-time verification, complemented by dynamic type checking at runtime [134].

In the following text, a brief overview of PHP's types and object-oriented system will be presented. For a more comprehensive explanation, please refer to the official documentation [134].

Numeric Types. Numeric types in PHP include `int` and `float`, which represent signed integers of platform-dependent size (typically 32 or 64 bits) and floating-point numbers, respectively. The `float` type behaves as a "wider" type to `int` [134].

Enumeration Types. PHP has a boolean type `bool` as well as support for user-defined enumerations (`enum`) [134].

Subrange Types. PHP does not have subrange types.

Composite Types. Composite types in PHP include `array`, resource, and object [134]. The `array` is implemented as an ordered collection of key-value pairs where both key and value are heterogenous; such a data structure can behave like other container types, including lists, hash tables, or dictionaries [135]:

```php
<?php
$array = array(
    "foo",
    "bar",
    "hello" => "world",
    -100,
);
var_dump($array);

// array(4) {
//   [0]=>
//   string(3) "foo"
//   [1]=>
//   string(3) "bar"
//   ["hello"]=>
//   string(5) "world"
//   [2]=>
//   int(-100)
// }
```

The resource is a special variable that refers to an external file, database connection, image canvas area, etc.

Type `string` is a special case as it can behave as both composite [136] and numeric [137] type. It represents a series of bytes where a byte can represent either a letter or a digit. This leads to interesting behavior like [137]:

```php
$foo = 1 + "10";   // $foo is 11
$foo = 1 + "10.5"  // $foo is 11.5
$foo = 1 + "bob"   // TypeError
```

The Listing 9 demonstrates some key features of PHP's object-oriented system. Lines 2–4 define the interface `Greetable`, which requires the `greet` method to be implemented by any class that implements it. On line 6, the abstract class `A` is declared, which implements the interface. This class cannot be instantiated due to its abstract nature. The class attribute `$privateAttr` (line 7) is private, meaning it can be accessed only within the scope of the class. The method `privateGreet()` (lines 9–11) is also private, meaning it can be called within the class only, while the protected method `protectedGreet()` (lines 13–15) can also be called from any class that inherits from `A`. Lastly, the method `greet` (line 17) is both abstract—requiring any subclass to provide an implementation—and public, meaning that it can be called outside the class `A`.

Line 20 displays simple inheritance (PHP does not support multiple inheritance). The function `say_hi` (lines 35–37) accepts any class implementing the interface `Greetable`, with lines 41 and 42 demonstrating how this can bring parametric polymorphism.

The provided example demonstrates several aspects of PHP's object-oriented system, including interfaces, abstract classes, inheritance, encapsulation, method visibility (public, private, protected), and polymorphism through the use of interfaces.

```php
<?php
interface Greetable {
    public function greet();
}

abstract class A implements Greetable {
    private $privateAttr = 'Hello';

    private function privateGreet() {
        echo $this->privateAttr . " World!\n";
    }

    protected function protectedGreet() {
        $this->privateGreet();
    }

    abstract public function greet();
}

class B extends A {
    public function greet() {
        $this->protectedGreet();
    }
}

$b = new B();
$b->greet();  // 'Hello World!'

class C extends A {
    public function greet() {
        echo "Bonjour Le Monde!\n";
    }
}

function say_hi(Greetable $x) {
    $x->greet();
}

$c = new C();

say_hi($b); // 'Hello World!'
say_hi($c); // 'Bonjour Le Monde!'
?>
```

Code listing 9 Demonstration of PHP object-oriented system

3.3.2 Towards Gradual Typing

The following text delves into the history of gradual typing in PHP, tracing the evolution of type annotations and the gradual typing system in the language. Unlike Python and Ruby, which embraced more gradual and community-driven approaches, PHP's path to gradual typing has been shaped by the practical needs of its user base and the language's specific design goals. Through key milestones and decisions, the PHP's gradual typing evolution will be explored.

In 2011, Facebook announced the Hip Hop Virtual Machine (HHVM), an open-source virtual machine specifically designed for executing PHP code [138]. HHVM was developed to speed up the development process and to replace the older transpiler HipHop. The main advantage of HHVM is its ability to translate PHP code dynamically into native machine code.

In that time, Facebook's thousands of PHP developers faced significant challenges in their day-to-day work [139] One common bug involved calling a method on a null object, resulting in runtime errors [139]. Furthermore, the complex API required developers to frequently consult documentation [139] (note the similarity to the Subsection 3.2.3).

In 2012, Julien Verlauget and Alok Menghrajani, two Facebook developers, started working on a solution. Their initial goal was to run a static analysis to identify and address vulnerabilities in the code. As they collaborated with the HHVM team and expanded their own team, they incrementally developed a more effective tool. This bottom-up approach proved successful, as other engineers appreciated the improvements and provided valuable feedback. Consequently, the tool gained widespread adoption within the company [140].

In 2014, three years after the introduction of HHVM, the tool was unveiled as Hack – a new dialect of PHP designed primarily for use with HHVM [139]. Hack was developed as a gradually typed and interoperable language with PHP, aiming to address prevalent issues in PHP programs and offer a more sophisticated API while maintaining the capacity for rapid iteration [139].

It was designed to bring the best of both worlds (static and dynamic) and one mental model (if you know PHP, you know Hack) [140]. Although most PHP files were valid Hack files, Hack did not support certain PHP features and introduced new ones. One of the most significant features introduced by Hack was the ability to add type annotations to function signatures and class members. This allowed developers to specify the types of variables and function return values, resulting in a more robust codebase. For instance:

```
<?hh // strict
function add(int $a, int $b): int {
    return $a + $b;
}
```

In the example above, first notice that instead of the `<?php` tag, the `<?hh` tag is used signaling Hack is used. Follows the `// strict` comment telling the static type checker, it should type-check all language constructs – other options are `// decl` (only interfaces are checked) and the default option that assumes that the code is correct if an annotation is missing and that unknown functions and classes are defined in PHP (even when they are not) [140]. The function `add` takes two integer parameters and returns an integer.

Hack also provided a new set of container classes called Collections, which were designed to be more efficient and safer than PHP's built-in arrays [140]. Some examples of Hack collections are `Vector`, `Map`, and `Set`:

```
$vector = new Vector<int>(1, 2, 3);
$map = new Map<string, int>('a' => 1, 'b' => 2);
$set = new Set<string>('apple', 'banana', 'orange');
```

These Collections have more predictable behavior and better performance than PHP arrays, as they are strongly typed and do not rely on key-value pairs. They also have their immutable

variants `ImmVector<T>`, `ImmMap<Tk, Tv>` and `ImmSet<T>`. Generic classes and nullable types would behave as follows [140]:

```
<?hh // strict
class Mailbox<T> {
    private ?T $data = null;

    public function send(T $data): void {
        $this->data = $data;
    }

    public function fetch(): ?T {
        return $this->data;
    }
}

$mailbox = new Mailbox<string>();
$mailbox->send('Hi!');
$msg = $mailbox->fetch();
if ($msg !== null) {
    echo $msg; // Hi!
}
```

In this example, `Mailbox` is a generic class with a type parameter `T`. The `?` before the parameter `T` denotes that the variable `data` can be either `null` or of type `T`. The *send* method accepts a value of type `T` (never `null`) and does not return anything, while the *fetch* method does not take any parameters and returns a nullable type `T`.

Apart from the type-specific changes, Hack introduced other benefits such as more concise syntax for lambda expressions and constructors, as well as more consistent behavior for trailing commas (making them always optional) [140]. While these changes were not necessarily directly connected to gradual typing, they did have interesting effects. The concise syntax might have offset the extra code introduced by the type annotations. Improved syntax overall might have helped boost adoption and overcome initial doubts within the community.

Hack offered IDE support with fast execution, rapid type checking, and autocomplete [140]. The Hack type checker was designed as a server and functioned like a daemon, listening to kernel events. It tracked dependencies and re-computed when a file changed [140].

The Facebook team presented their work on Hack at the Hack Developer Day 2014 event. Shortly after, the Hack transpiler (h2tp) was introduced as a simple command-line tool. It allowed projects converted from PHP to Hack to continue making releases targeting the PHP language [141]. The tool provided forward compatibility with PHP by removing type annotations and converting collections and lambda expressions into PHP constructs [141]. Some challenges in the implementation of h2tp included the reference semantics of Hack collections and the difference in handling empty collections between Hack and PHP [141].

In 2015, several large organizations, including Box, Etsy, and Wikipedia, announced their adoption of HHVM [142]. While these companies switched to the virtual machine rather than the Hack language, the adoption of HHVM demonstrates the potential for widespread adoption of Hack in the future. This growing interest in type systems and the success of Hack may have also contributed to the inclusion of static types in PHP itself.

3.3.3 Emergence of Static Types in PHP

As a response to the growing interest in type systems, the release of PHP 7 in 2015 marked the beginning of the introduction of static types in PHP. This release included support for return type declarations and scalar type declarations [143, 144]. Return type declarations allowed specifying the type of value that a function should return, which could be a scalar type (`int`, `float`, `bool`, and `string`), a class, an interface, or the `self` keyword:

```
function test(): int {}
```

Scalar type declarations enabled annotating the types of function or method parameters:

```
function test(bool $param) {}
```

The official reasons for adding these features were to validate the programmer's intent and to allow for static analysis and static error detection [144].

With the release of PHP 7.1 in 2016, nullable types, the `void` type, and the iterable pseudo-type were introduced [143, 145]. Nullable types followed the same syntax as in Hack:

```
declare(strict_types=1);

function foo(): ?int {
return null;
}
```

Notice the directive on the first line enforcing strict type checking – that means only a value corresponding exactly to the type declaration will be accepted on function or method call or as a return value. This directive is valid for the given file only.

The iterable pseudo-type acted as a built-in compile-time type alias for `array | Traversable`, allowing for static analysis on any array-like structures.

PHP 7.2, released in 2017, added support for the `object` type [143, 145]:

```
function test(object $obj) : object {
return new SplQueue();
}

test(new stdClass());
```

In this example, the function `test` accepts a parameter of type object and returns a value of the same type.

PHP 8.0, released in 2020, introduced union types and the `mixed` type [143, 145]. Unlike Hack "union" types, PHP allows the programmer to combine any built-in types or classes:

```
function foo(): X|Y
```

The `mixed` type is a supertype of all other types and is functionally equivalent to not specifying a type. Semantically, it informs the reader that the generic type was intentionally chosen rather than omitted:

```
function foo(mixed $X): mixed
```

Finally, PHP 8.1, released in 2021, added support for intersection types [143, 145]:

```
function foo(): X&Y
```

In this case, the return value must be an instance of a class that is a subtype of both types `X` and `Y`.

3.3.4 Typed PHP in 2023

Throughout this chapter, PHP and its dialect Hack have been discussed. In the previous two subsections, it was shown that while Hack led the way with the adoption of many gradual-typing features, at the moment both are offering good support for writing both statically and dynamically typed code. As PHP is the more popular of the two (ranking 9th versus 50–100 on the TIOBE rankings [11]), it will be used for all demonstrations below.

To gain a better understanding of the current adoption of gradual typing, the 10 most-starred PHP repositories were reviewed. For each repository, contribution guidelines and the main file were examined. Table 3.2 indicates whether inline annotations, PHPDoc annotations combined with a static type checker (see Subsection 3.3.5), or both were used. For the purposes of this statistic, any of these approaches count as gradual typing. Out of the top 10 most popular repositories, 8 are using some form of gradual typing; 6 of them are employing inline annotations as discussed in Subsection 3.3.3, 4 in combination with one of the static analysis tools introduced in Subsection 3.3.5. Laravel, one of these repositories, is currently transitioning from PHPDoc to inline annotations only [146]. Out of the 2 repositories without any notion of gradual typing, one has been archived since 2021. This paints a clear picture: in the world of PHP, major tools are embracing gradual typing. To expand on this observation, a more comprehensive analysis beyond the scope of this thesis would be necessary.

Table 3.2 Overview of type annotations in popular PHP repositories

Repository Name	Stars	Annotated	Comment
Laravel [147]	73.3k	Yes	Moving towards in-line only.
jQuery-File-Upload [148]	31.0k	No	Repository archived in 2021.
The Laravel Framework [149]	29.4k	Yes	PHPDoc / PHPStan
Symfony [150]	28.3k	Yes	In-line + PHPDoc
Composer [151]	27.6k	Yes	In-line + PHPDoc / PHPStan
Faker [152]	26.7k	Yes	PHPDoc. Repository archived in 2020.
Guzzle [153]	22.5k	Yes	In-line + PHPDoc / PHPStan
NextCloud [154]	22.4k	No	
DesignPatternsPHP [155]	21.2k	Yes	In-line
Monolog [156]	20.4k	Yes	In-line + PHPDoc / PHPStan

The Listing 10 demonstrates how the same code can be written in PHP both with (right) and without (left) annotations:

- Line 3 specifies that types of passed values need to match exactly to the types on annotations
- The constructor on line 7 includes type annotations for both input parameters and the return type. This makes it clear that the constructor accepts two arguments of either int or float and returns nothing.
- The `add` method, starting on line 17, showcases the use of Union types. The type annotation for the argument, specified on line 18, demonstrates that it can be either an instance of the Vec2D class or a scalar value (int or float).

- The `add` method, on lines 17–31, performs different operations depending on the type of input. If the other argument is an instance of Vec2D, it performs vector addition. In the case of scalars (int or float), it first converts them into a Vec2D instance by setting both the x and y components to the scalar value and then performs the addition.

```php
<?php
// Untyped code




class Vec2D {
    public $x;
    public $y;

    public function __construct($x, $y) {
        $this->x = $x;
        $this->y = $y;
    }




    public function add($other) {
        if ($other instanceof Vec2D) {
            return new Vec2D(
                $this->x + $other->x,
                $this->y + $other->y
            );
        } else {
            return new Vec2D(
                $this->x + $other,
                $this->y + $other
            );
        }
    }




    public function __toString() {
        return "({$this->x}, {$this->y})";
    }
}

$v1 = new Vec2D(1, 2);
$v2 = new Vec2D(3, 4);
$v3 = new Vec2D(1.5, 2.5);

echo $v1->add($v2) . "\n"; // Out: (4, 6)
echo $v1->add($v3) . "\n"; // Out: (2.5, 4.5)
echo $v1->add(2) . "\n";   // Out: (3, 4)
```

```php
<?php
// Typed code

declare(strict_types=1);

class Vec2D {
    public float|int $x;
    public float|int $y;

    public function __construct(
        float|int $x, float|int $y
    ) {
        $this->x = $x;
        $this->y = $y;
    }

    public function add(
        Vec2D|float|int $other
    ): Vec2D {
        if ($other instanceof Vec2D) {
            return new Vec2D(
                $this->x + $other->x,
                $this->y + $other->y
            );
        } else {
            return new Vec2D(
                $this->x + $other,
                $this->y + $other
            );
        }
    }

    public function __toString(): string {
        return "({$this->x}, {$this->y})";
    }
}

$v1 = new Vec2D(1, 2);
$v2 = new Vec2D(3, 4);
$v3 = new Vec2D(1.5, 2.5);

echo $v1->add($v2) . "\n"; // Out: (4, 6)
echo $v1->add($v3) . "\n"; // Out: (2.5, 4.5)
echo $v1->add(2) . "\n";   // Out: (3, 4)
```

Code listing 10 Comparison of PHP code without and with type annotations

3.3.5 Tooling

Unlike Ruby or Python, PHP has both type definitions and type checking built into the interpreter. This all-in-one approach decreases the need for tooling supporting gradual typing. However, there are still some tools available to enhance the gradual typing experience in PHP.

PHPStan. PHPStan is a static analysis tool that works best with annotated PHP code [157]. Table 3.2 demonstrates that at least among the most popular GitHub repositories, PHPStan is the number one tool for static analysis. It uses the documentation-in-comment format PHPDoc for adding more type-related information and for using features not yet supported by the language, such as generic classes. An example of PHPStan usage:

```
/** @param array<int, Item> $items */
function foo(array $items) {
...
}
```

Psalm. Psalm is another static analysis tool that, similar to PHPStan, uses PHPDoc for annotations [158]. It supports various type annotations, including variable types, function and method return types, function and method parameter types, closures, type aliases, import types, and union types.

Phan. Phan is a static analyzer for PHP that aims to prove incorrectness rather than correctness [159]. It also uses PHPDoc and checks for type safety. Phan supports various type annotations, including methods, functions, closures, union types, generic types, and generic arrays (e.g., `int[]`, `UserObject[]`, `array{key:string,value:?stdClass}`, etc.).

Rector. Rector is a tool for instant upgrades and automatic refactoring [160]. It is capable of instantly moving codebases between PHP versions and performing automatic refactoring. Rector specializes in achieving type coverage, which is a metric with the following formula [161]:

$$\frac{\text{Number of type annotations}}{\text{Number of possible type annotations}}$$

In summary, while PHP has both type definitions and type checking integrated into the interpreter, reducing the need for additional tooling to support gradual typing, several tools are available to enhance the gradual typing experience in PHP.

3.3.6 Evaluating PHP's Type System

PHP unlike Python or Ruby does have some type checks built into the language. This is why, within the rest of the text, it will be discussed on its own without considering any of the above-mentioned static type checkers. On the one hand, this might be looked at as an unfair comparison. On the other hand, it allows the reader to compare three distinct approaches: static checks only (Python with mypy), a combination of static and dynamic checks (Ruby with Sorbet), and dynamic checks only (PHP).

Type Soundness. To the author's knowledge, no proof of PHP's soundness was published and therefore will be considered type unsound.

Type-Enforcement Strategy. Out of the type enforcement strategies described by Greenman et al. [24], PHP is closest to the transient strategy as it does type-checking during runtime but does not type-check untyped code whatsoever.

In conclusion, PHP's type system can be considered possibly type safe but not type sound while using the transient type-enforcement strategy.

3.3.7 Summary

In this section, the gradual typing in PHP was reviewed. The development of gradual typing in PHP was strongly influenced by Facebook and their creation of the Hack language, a variant of PHP. Eventually, the core PHP language caught up, and both PHP and Hack now offer competitive gradual typing features.

It is intriguing to observe two closely related languages implementing gradual typing simultaneously and the distinct decisions each language made. While Hack began with a comprehensive set of gradual typing features, PHP added support incrementally, starting with method annotations and basic built-in types. Subsequently, PHP introduced nullable types, `void` and `iterable`, followed by `object`, `union`, `mixed`, and, most recently, intersection types.

Some noteworthy differences between PHP and Hack include Hack's support for generic classes, which PHP lacks, and PHP's support for union types in a similar way to Python and Ruby (`X|Y`), which Hack does not offer. These distinctions showcase how languages with similar foundations can develop and adopt gradual typing in unique ways, resulting in diverse feature sets and implementation strategies.

PHP's implementation of gradual typing is also distinct from other languages. Type definitions and type checking are integrated into newer versions of PHP, as opposed to being separate tools. This integration further highlights the unique approach PHP has taken in developing and implementing gradual typing.

3.4 Comparison

As discussed throughout this thesis, Python, Ruby, and PHP are three popular programming languages, each with its specific strengths and use cases. Python, as outlined in Section 3.1, is a versatile language with applications ranging from web development to business applications and scientific computing. Ruby, as presented in Section 3.2, and PHP, as detailed in Section 3.3, are both particularly popular for web development.

These three languages share several common features. They are all multi-paradigm languages, meaning they support various programming styles, such as procedural, object-oriented, and functional programming. Furthermore, they each possess a mature object-oriented system, making them well-suited for large-scale software projects.

All three languages are open-source, which encourages collaboration and contributions from an active community of developers. This collaborative environment has led to the continuous improvement and evolution of these languages over time. Until recently, Python, Ruby, and PHP were predominantly dynamically typed. However, with the advent of gradual typing, each of these languages has incorporated optional static typing to improve type safety and enable better development tooling.

3.4.1 Implementation Strategy

All three languages are following path of the static retrofitting – that is adding static types to otherwise dynamically typed language. For each of them, however, the initial impulse came from a slightly different direction.

For Python, the gradual typing journey began with the language's author, Guido Van Rossum, who outlined the theoretical foundation. Then syntax for type annotations was added, followed by the emergence of the popular type checker, mypy. After that, semantics were added to the core language. Large tech companies recently created their own static type checkers, built on the foundations set by the core development team. Currently, gradual typing in Python continues to develop with a strong roster of external tools employing the standards set by the core development team.

In the case of Ruby, the journey started in academia with Diamondback Ruby. After several years of development, the commercial sphere joined in. Both Sorbet and Steep were built on top of the work done by academia, even adapting their type definitions. The core language started its gradual typing journey by creating RBS, a special language for Ruby type annotations. While academic research continues, Sorbet and Steep are currently the only two relevant options for practical use.

As for PHP, the journey began with Hack, a dialect developed by Facebook. The tech giant developed Hack for their own needs and released it to the world. The core language subsequently started to add its own gradual typing features. Unlike Ruby and Python, runtime type checking was a built-in part of the language from the beginning of this effort. External static type checkers were developed in parallel with the core language, often adding features not yet supported by the core language. Both Hack and PHP are actively developed alongside their ecosystems, allowing developers to choose based on their preferences.

Table 3.3 Comparison of Implementation Strategy

Strategic Aspect	Python (mypy)	Ruby (Sorbet)	PHP
Approach	Static Retrofitting	Static Retrofitting	Static Retrofitting
Initiator	Core Dev Team	Academic	Commercial
Language Syntax	Extend	Exploit	Extend
Type Checking Time	Compile Time	Both	Runtime
Type Checking Responsibility	External	External	Both
Type Definitions Repository	Typeshed	RBI Central	None

As can be seen in Table 3.3, each of the languages opted for a slightly different implementation strategy. Python and PHP extended existing language syntax allowing for more concise and readable code, while Ruby did not change its syntax keeping annotation in a separate file, making it easy to opt in and out from static typing without any change to the source file. While mypy checks during the compile time, Sorbet does both compile time and runtime checks allowing the user to opt-out from the runtime ones [162] and PHP itself provides runtime checks with compile time checking provided by external type checkers. Both Python and Sorbet do have a central repository with type definitions for standard library as well as external libraries – PHP on the other hand does not have anything like that. Standard library type definitions are built in the language while type definitions for external libraries type definitions are either shipped with the library or can be found in separate packages.

3.4.2 Syntax

Table 3.4 highlights the similarities and differences among the three languages regarding their syntactic features. The position of type annotations within the code varies across the three languages. While both PHP and Python have integrated type annotations into their language syntax, Ruby has taken a different approach. Ruby does not have native language support for type annotations, except for the RBS language which allows specifying annotations in a separate file. Consequently, Sorbet developers had to use special methods for type annotations. The advantage of this approach lies in excellent compatibility with older code and other tools. However, the downside is a more verbose syntax compared to the other two languages.

After introducing type annotations to a language, not all language features are typically possible or practical to annotate. In such cases, it might be beneficial to leave some parts of the code intentionally dynamically typed. All three languages provide syntax for this type of interaction, such as the `any` keyword. This type acts like a supertype of all other types, allowing developers to opt out of static checks. It is particularly useful when annotating a codebase,

Table 3.4 Comparison of Syntactic Features

Syntax Feature	Python (mypy)	Ruby (Sorbet)	PHP
Annotations – parameters	Inline (after)	`sig` method (above)	Inline (before)
Annotations – return value	Inline (after)	`sig` method (above)	Inline (after)
Annotations – variables	Inline (after)	Inline (`let` method)	Inline (before)
Universal Supertype	`Any`	`T.untyped`	`mixed`
Nullable Types	`Optional[int]`	`T.Nilable[Integer]`	`?int`
Type aliasing	Yes	Yes	Yes
Union types	Yes	Yes	Yes
Intersection types	Yes	Yes	Yes
Type inference	Yes	Yes	Yes

as it helps distinguish between intentionally dynamically typed expressions and those not yet annotated.

In dynamically typed languages, all types are typically nullable, with NULL often used as a sentinel value [8]. However, in many cases, NULL is not needed. This is why the types in all three gradual type systems are non-nullable by default, providing additional syntax for marking a type as nullable when necessary.

The common features shared by all three languages also include type aliasing, union types, and intersection types. These features contribute to a more concise syntax and facilitate a better programming experience. Type inference is another universally embraced feature, present in all PHP, Sorbet, and mypy.

3.4.3 Semantics

Table 3.5 summarizes the similarities and differences among the three languages regarding concepts related to type soundness, and polymorphism.

Table 3.5 Comparison of Semantic Features

Semantic Feature	Python (mypy)	Ruby (Sorbet)	PHP
Type Sound	No	No	No
Optional Strictness	No	Yes	Yes
Type-Enforcement Strategy	Erasure	Hybrid	Hybrid
Parametric Polymorphism	Generics	Type Variables	Generics
Nominal Subtyping	Yes	Yes	Yes
Structural Subtyping	Protocols	Interface Types	No
Type Bounds	Yes	Yes	Yes
Covariance / Contravariance	Yes	No	No
Method Overloading	No	No	Yes

The three type systems exhibit minor differences in terms of soundness. While each of the type systems displays some level of unsound behaviors [128], Sorbet unlike the remaining two performs both runtime and compile time run checks and behaves soundly in most cases.

Each language employs a different type-enforcement strategy, as discussed in Subsection 2.2.3. Python uses erasure, which offers runtime speed at the cost of soundness. Sorbet and PHP provide configurable levels of strictness, allowing programmers to balance the strength of checks and the runtime performance of the program. As a result, their type-enforcement strategy can

be considered hybrid, with Sorbet leaning more towards natural enforcement and PHP leaning more towards transient enforcement.

All three languages support nominal subtyping and the ability to specify type bounds. Although Sorbet employs a different approach to parametric polymorphism, all three languages offer the same basic functionality.

Both Python and Ruby provide their own flavor of interfaces to offer a statically typed alternative to duck typing. While PHP also supports interfaces, they are based on nominal subtyping, requiring an explicit implementation of an interface for type compatibility.

While all three languages exhibit covariant and contravariant behavior, only Python enables developers to specify which behavior should apply in a given context. Finally, PHP is the only language that allows for method overloading.

3.5 Other Gradually Typed Languages

The 3 languages were chosen for their popularity, parallels to the R language, and diversity, but there are many more. For example, TypeScript, Typed Racket, and Typed Lua.

TypeScript, developed by Microsoft, is a gradually typed language based on JavaScript [163]. It has gained significant popularity and widespread adoption, ranking 4th on GitHub's list of most-used programming languages in 2022 [9], 5th in the programming, scripting, and markup languages category of Stack Overflow's 2022 Developer Survey [10], and 40th on the TIOBE Programming Community index for April 2023 [11].

Typed Racket is another gradually typed language, which serves as a sister language to the Racket programming language [164]. Although not explicitly mentioned on the language leaderboard, it has been used and iterated on in academia, providing valuable insights into the development of gradually typed languages.

Typed Lua, introduced by Maidl et al. in 2014, is a dynamically typed extension of the Lua programming language [165]. It offers a novel approach to statically type Lua's unconventional metaprogramming features, further expanding the possibilities for gradual typing. Like Typed Racket, Typed Lua is not explicitly mentioned on the language leaderboard, but its unique characteristics make it an interesting case study in the context of gradually typed languages.

Chapter 4

Towards gradually typed R

This chapter introduces the R programming language and discusses its need for gradual typing. It also discusses various approaches for implementing gradual typing in R, taking into consideration the insights gained from the previous sections.

4.1 R Programming Language

The R programming language is an open-source language and environment primarily designed for statistical computing and graphics [166]. R was developed as a dialect of the S language and environment, which was initially created at Bell Laboratories (formerly AT&T, now Lucent Technologies) by John Chambers and colleagues [166]. Given its focus on statistical and graphical applications, R can be considered a domain-specific language (DSL) for statisticians.

The philosophy behind R is to provide an easy-to-use and extensible environment, with well-designed outputs and thoughtful defaults [166]. It supports mathematical notation as output, making it highly suitable for academic and research applications. The R environment is an integrated suite that offers effective data handling and a large collection of tools for data analysis [166]. It can also be extended with packages via the Comprehensive R Archive Network (CRAN) [166]. R even has its Markdown-like documentation format [167].

According to Morandat et al. [6], the R language can be described as a multi-paradigm, supporting functional, object-oriented, and imperative programming styles. It is dynamic, allowing users to modify existing code and create new functions at runtime, and interpreted, as it uses an abstract syntax tree representation for code execution. Similarly to Python and Ruby, it features duck-typing.

The R programming language is currently supported by the R Foundation, a non-profit organization founded by members of the R Development Core Team [168]. This foundation helps to manage and promote the R language and its community.

In terms of popularity, R did not make the top 10 most popular languages on GitHub's 2022 list [9]. However, it ranked 20th in the programming, scripting, and markup languages category in Stack Overflow's 2022 Developer Survey [10] and 16th on the TIOBE Programming Community index for April 2023 [11]. These rankings indicate that R remains a widely-used language, particularly within the fields of statistics, data analysis, and scientific research.

To summarize, R is an open-source, domain-specific programming language focused on statistical computing and graphics. With its rich ecosystem, extensibility, and support for various programming paradigms, R continues to be a popular choice for researchers and data analysts.

4.1.1 R Built-in Types

R offers a diverse range of built-in types [169], several of which are discussed in this section. Notably, R's primitive types are vectorized, meaning that scalars are simply unit vectors [7]. Although there are no explicit types in R, internally the R interpreter works with the following RunTime Type Information (RTTI):

Numeric Types. Numeric types in R feature `integer`, `double`, `complex` – a vector of integers, vector of floating point numbers, and vector of complex numbers respectively. The coercion rules are not global. They are local to each function and follow a convention rather than specification. That being said, the usual hierarchy goes as follows: `logical` is a narrower type than `integer` which is a narrower type than `double` which is a narrower type than `complex` [170].

Enumaration Types. Enumeration types in R include `logical` and `factor`. `logical` is a vector of boolean values that are unusual in that it apart from `TRUE` and `FALSE` also includes `NA` – a value that represents missing data. `factor` is a special compound type used to describe limited sets of categorical data – in principle similar to enums in C-based languages.

Subrange Types. There are no subrange types in R.

Composite Types. Composite types in R encompass a variety of data structures, including:

- `character` is a vector of text strings.
- `list` is a heterogenous vector of R object, including other lists, vectors, or data frames.
- `raw` is a homogenous vector of raw bytes.

Apart from the types above, R does have other two special types of containers `data.frame` and `matrix`. `data.frame` is a list of vectors, factors, and/or matrices where each of the vectors has the same length. In essence, it is a two-dimensional table where columns represent variables and rows represent observations. Data frames are a fundamental data structure in R for handling tabular data, as they can store heterogeneous data types and can be easily manipulated, aggregated, and analyzed. `matrix` is a two-dimensional array that can store homogeneous data types, such as numeric or character values. Matrices in R are created with the `matrix()` function and their elements can be accessed or modified using row and column indices. Matrices are useful for linear algebra operations, as well as for organizing and manipulating large sets of homogeneous data.

NA and NULL. Given the nature of the language, the expression `NA` and `NULL` deserve a special mention. While `NULL` is its own singleton type used for indicating that variable has not been assigned value or that function does not return a meaningful value, `NA` represents missing or unavailable data [169]. `NA` can be an instance of other data types such as integer, double, or logical [169]. For a language that is often used for analyzing real-world data, understanding this distinction is crucial.

4.1.2 Object Oriented Systems

This subsection draws on *Advanced R* by Hadley Wickham [171, Chapter 3 Data structures], with original code snippets illustrating the concepts. Unlike most other programming languages, R has 3 Object Oriented systems dubbed *S3*, *S4*, and *R6*. These systems are independent of each other and – while not advisable – it is possible to use them together. The following text will briefly demonstrate the common tasks: the creation of new classes, workarounds for subtyping, and method manipulation during runtime.

S3 Object-Oriented System. Listing 11 shows the S3 system. It relies on generic functions and method dispatch based on the class of the first argument, which allows for easier code extension and modification. The S3 system doesn't enforce strict class definitions, and it doesn't require the specification of method signatures. This can lead to less robust and maintainable code compared to the S4 system.

The initial 11 lines define constructor functions for new classes, `A` and `B`, each with one attribute. Although traditional subtyping is not supported, workarounds like the one demonstrated on lines 13–20 are available. Lines 22–31 highlight the addition and usage of methods `greet_A` and `greet_B` for an existing class. Line 33 exhibits the removal of a method during runtime, preventing further calls to the method. The `say_hi()` function (line 38) accepts an object and calls its `greetA()` method, illustrating duck-typing in action. Finally, lines 40–41 show the `say_hi()` function working with instances of classes `A` and `SubAB`, regardless of their actual types.

S4 Object-Oriented System. Listing 12 demonstrates the S4 system. Similarly to S3, it is copy-on-write. In contrast to S3, it provides a more formal approach to object-oriented programming. It introduces a stricter class definition and enforces the use of methods with signature specification, which makes the code more robust and maintainable. The S4 system allows for multiple inheritance and provides tools for method dispatch based on the classes of multiple arguments.

Here, classes `A`, `B` and `SubAB` are defined using the `setClass()` function (lines 1–3), with constructor functions for `A` and `SubAB`. Notice that class `SubAB` inherits from both `A` and `B`. Methods are created and added for existing classes using the `setGeneric` and `setMethod()` functions (lines 8–14). The `say_hi()` function (line 22) accepts an object and calls its `greetA()` method, illustrating duck-typing in action. Finally, lines 24–25 show the `say_hi()` function working with instances of classes `A` and `SubAB`, regardless of their actual types.

Overall, the S4 system provides more structure and formality than the S3 system, offering stricter class definitions, support for multiple inheritance, and method dispatch based on the classes of multiple arguments.

R6 Object-Oriented System. Listing 13 demonstrates the R6 system. In contrast to the S3 and S4 systems, the R6 system utilizes classes and methods that are explicitly defined within a single R6Class object. Additionally, R6 supports private methods and fields, which can improve encapsulation and modularity in the code. While R6 may be more complex than the S3 system, its reference-based semantics allowing objects to be modified in place and modern features make it an attractive option for larger projects and developers familiar with other object-oriented languages.

During the definition of the class `6` (lines 3–8) both its attribute `AttrA` and method `greet` are defined. Lines 10–19 display single inheritance as class `SubB` inherits from `B`. Multiple inheritance is not supported. Classes are instantiated using method new on lines 21–22. Lastly, lines 26–29 are demonstrating that even the R6 system still supports duck-typing.

This example demonstrates that the R6 system is the most similar to modern Object-Oriented systems as seen in languages such as Python (Listing 1) or Ruby (Listing 7) programming languages.

```
create_A <- function() {
  A <- list(AttrA = "Hello")
  class(A) <- "A"
  return(A)
}

create_B <- function() {
  B <- list(AttrB = "Bonjour")
  class(B) <- "B"
  return(B)
}

create_SubAB <- function() {
  A <- create_A()
  B <- create_B()
  structure(
    list(AttrA = A$AttrA, AttrB = B$AttrB),
    class = c('SubAB', 'B', 'A')
  )
}

greet_A <- function(object) UseMethod("greet_A")
greet_A.SubAB <- function(object) cat(paste(object$AttrA, "World!", sep = ' '))

greet_B <- function(object) UseMethod("greet_B")
greet_B.SubAB <- function(object) cat(paste(object$AttrB, "Le Monde!", sep = ' '))

subAB <- create_SubAB()

greet_A(subAB) # "Hello World!"
greet_B(subAB) # "Bonjour Le Monde!"

rm(greet_B.SubAB)

a <-create_A()
greet_A.A <- function(object) cat(paste(object$AttrA, "Welt!", sep = ' '))

say_hi <- function(object) greet_A(object)

say_hi(a) # "Hello Welt!"
say_hi(subAB) # "Hello World!
```

Code listing 11 Demonstration of R S3 object-oriented system

```r
setClass("A", representation(AttrA = "character"), prototype(AttrA = "Hello"))
setClass("B", representation(AttrB = "character"), prototype(AttrB = "Bonjour"))
setClass("SubAB", contains = c("B", "A"))

create_A <- function() new("A")
create_SubAB <- function() new("SubAB")

setGeneric("greet_A", function(object) standardGeneric("greet_A"))
setMethod("greet_A", "A",
          function(object) cat(paste(object@AttrA, "World!", sep = ' ')))

setGeneric("greet_B", function(object) standardGeneric("greet_B"))
setMethod("greet_B", "SubAB",
          function(object) cat(paste(object@AttrB, "Le Monde!", sep = ' ')))

a<- create_A()
subAB <- create_SubAB()

greet_A(subAB) # "Hello World!"
greet_B(subAB) # "Bonjour Le Monde!"

say_hi <- function(object) greet_A(object)

say_hi(a) # "Hello World!"
say_hi(subAB) # "Hello World!"
```

Code listing 12 Demonstration of R S4 object-oriented system

```
library(R6)

A <- R6Class("A",
  public = list(
    AttrA = "Hello",
    initialize = function() self$AttrA <- "Hello",
    greet = function() cat(paste(self$AttrA, "World!", sep = ' '))
))

B <- R6Class("B",
  public = list(
    AttrB = "Bonjour",
    initialize = function() self$AttrB <- "Bonjour"
))

SubB <- R6Class("SubB", inherit = B, public = list(
  initialize = function() super$initialize(),
  greet = function() cat(paste(self$AttrB, "Le Monde!", sep = ' '))
))

a <- A$new()
subB <- SubB$new()

subB$greet() # "Bonjour Le Monde!"

say_hi <- function(object) object$greet()

say_hi(a) # "Hello World!"
say_hi(subB) # "Bonjour Le Monde!"
```

Code listing 13 Demonstration of R6 object-oriented system

4.1.3 Type Operations

The R programming language offers a set of tools that allow developers to work around the absence of type annotations and ensure that the correct type is provided [172, 173, 174, 175, 176, 177, 178, 179, 170]. These tools might be useful when prototyping various features connected with adding gradual typing to the language.

Type Testing. R provides several type testing functions to determine the types of objects and values [172]. One such function is `typeof`, which determines the R internal type or storage mode. It returns values for types as described earlier, along with several other less common or internal types. For a full list of return values, refer to the documentation [172].

The `is.numeric` function is an internal generic primitive function that returns true if the base type of the class is double or integer (but not complex) and the value can reasonably be regarded as numeric (supports arithmetic and comparison operations) [173].

The `is.double` function returns true if its argument is of type double [174], and `is.integer` returns true if its argument is of type integer [175]. Similarly, `is.complex` returns true if its argument is of type complex [176], `is.logical` returns true if its argument is of type logical `r_logical`, and `is.character` returns true if its argument is of type character [178].

The `is.list` function returns true if the argument is a list with a length greater than 0 [179]. The `is.matrix` function, on the other hand, returns true if its argument is a vector with a "dim" (dimensions) attribute of length 2 [170]. Note that a data frame is not considered a matrix by this test.

Type Conversions. Type conversion functions in R are designed to change the type of an object explicitly. Some of the commonly used type conversion functions include `as.numeric`[173], `as.double`[174], `as.integer`[175], `as.complex`[176], `as.logical`[177], and `as.character`[178]. These conversion functions can be useful when working with gradual typing, as they allow developers to explicitly specify the desired type for a particular value or object.

As gradual typing is introduced to the language, this methods can be used for prototyping purposes as well as for understanding how the developers of the language are thinking about the shapes of values in the language.

4.2 Benefits of Gradual Typing in R

Like in the case of the programming languages discussed throughout Chapter 3, there is a wide range of ways in which the R programming language could potentially benefit from introducing optional static types. Most notably reliability, developer experience, documentation and potential for runtime performance optimizations.

Reliability. While the common interactive use of R is clearly a use case for dynamically typed code, adding types might come in handy once the analysis is done and the author wants to double-check the data before presenting it. After all, many data-driven decisions, especially in areas like finance, policy, public health, or healthcare, might be a matter of life and death! An even stronger case might be libraries. The R community is built on a foundation of a plethora of statistical packages [180]. Having a bug or error in one of them might wreak havoc through many reports and dashboards that utilize it.

Developer Experience. Gradual typing can provide numerous benefits, not only in terms of reliability but also in enhancing the development experience. Adding type annotations can significantly improve the developer experience within an Integrated Development Environment (IDE). As the IDE understands the types, it can suggest relevant completions for the code being written, allowing developers to quickly navigate to the definition of a particular function or variable by clicking on its usage. With type information available, IDEs can easily show all the places where a specific function or type is being used and display relevant documentation for functions and variables based on their types, making it easier to understand and use them. Moreover, with type information, developers can experiment with various code paths and receive immediate feedback, allowing them to iterate faster.

Documentation. Gradual typing can also serve as a form of documentation, making it easier for developers to understand how a function or library should be used. Type annotations can convey the expected input and output types, reducing the likelihood of misunderstandings and errors.

Performance Optimizations. The introduction of type annotations may allow for compiler or interpreter optimizations. With type information available, the compiler or interpreter can make better decisions regarding memory allocation and management, and even optimize vector operations, potentially leading to performance improvements.

Figure 4.1 Type Language for R [8]

$T ::=$	`any`	*top type*	$A ::= T$		*arguments*
|	`null`	*null type*	|	$\ldots$	*dots*
|	`env`	*environment type*	$V ::=$	S`[]`	*vector types*
|	S	*scalar type*	|	$\hat{}S$`[]`	*NA vector types*
|	V	*vector type*	$S ::=$	`int`	*integer*
|	$T \vert T$	*union type*	|	`chr`	*character*
|	$?T$	*nullable type*	|	`dbl`	*double*
|	$\langle A_1, \ldots, A_n \rangle \to T$	*function type*	|	`lgl`	*logical*
|	`list`$\langle T \rangle$	*list type*	|	`clx`	*complex*
|	`class`$\langle ID_1, \ldots, ID_n \rangle$	*class type*	|	`raw`	*raw*

In conclusion, the addition of gradual typing to R can greatly benefit the language and its users. It can improve the development experience, provide better documentation, and potentially enable optimizations that can enhance the performance of R programs. As R is widely used in critical domains, introducing gradual typing can contribute to more reliable and efficient data analysis and reporting.

4.3 Previous Work

Turcotte et al. [8] proposed a type language for R (Figure 4.1), which includes scalar types for annotating unit vectors, container types like list and vector, and several features previously discussed in this thesis such as function signatures, union types, nullable types, and a universal supertype. Additionally, they introduced ... argument types and NA-able types, which are particularly relevant in the R context.

To test their proposal, the authors used a type inference algorithm to annotate over 25,000 functions across 412 packages from R open-source libraries. They evaluated over 8,500 clients of these packages and end-user code from the Kaggle competition. Their findings showed that 97% of function parameters were NA-free, about 95% of function parameters could not be NULL, and roughly one-third of the parameters were treated as scalars rather than vectors or other types. Furthermore, they found that approximately 1.5% of function parameters were higher-order functions, and due to the complexity of R's object-oriented systems, they decided to leave type annotations of objects for future research. The study also identified that 10% of all class parameters of functions were matrices, while 8.15% were data frames.

Turcotte et al.'s results demonstrated that 80% of functions were monomorphic or had one polymorphic parameter, and 97.6% of parameter types and 87.7% of function types never failed. These findings indicated that their proposed type system provided a solid foundation for a future gradual type system.

Another related study by Wrenn et al. [181] focused on the static typing of vectors, arrays, and matrices. The authors used LiquidHaskell, a statically typed extension to the Haskell programming language, to model these R types.

Outputs of both of these works will be considered and extended in the subsequent section by proposing further improvements in implementation strategy, syntax, semantics, and adoption strategy.

4.4 Design Considerations

After a review of the gradual typing implementation in three dynamically typed languages and the current state of the type system in the R programming language, it is time to discuss the potential of adding gradual typing to R.

4.4.1 Implementation Strategy

Starting point is an existing language, which offers two implementation approaches: static retrofitting (adding static typing features to R), and natively gradual (building a new dialect of R from the ground up). Among the case studies discussed in this thesis, the Hack language (Subsection 3.3.3) was the only one to adopt the natively gradual approach. This choice, however, would require significant effort in developing not only the language itself but also the accompanying tooling. On the other hand, the static extension approach has proven successful in all the languages discussed, offering the advantage of utilizing the existing tooling and ecosystem and facilitating adoption since it doesn't require users to switch tools or learn a new (although very similar) programming language.

As demonstrated in Subsection 3.4.1, a gradual typing journey can be successful regardless of whether it is initiated by the core language development team, the academic sector, or the commercial sector. Any of them will like work just fine for R.

A crucial decision is whether to exploit or extend the existing syntax. The exploit approach, as exemplified by Sorbet, enables the development of a fully-featured (Subsection 3.2.5), performant, and popular (Subsection 3.2.6) type system, offering the advantage of smooth interoperability with existing tools and codebases. However, a comparison of Listing 7 and Listing 1 reveals that the extend approach leads to greater conciseness and readability. In the author's opinion, these benefits outweigh the drawbacks. It is recommended to employ the exploit technique for prototyping and to utilize the extend technique to drive adoption.

As for the timing of checks, Sorbet's approach, which involves both runtime and compile-time checking along with granular settings for the strength of these checks, appears to strike an optimal balance. This allows the programmer to find the right balance between type safety and type soundness on one hand, and flexibility and runtime performance on the other. The only obvious drawback here is the significant effort required to build such a robust system.

Considering that R is a GNU/free software project, the Python approach seems appropriate from the core language perspective. This involves developing syntax and semantics for a gradual type system (see Subsection 3.1.8) and allowing the community to develop tools that suit their needs (see Subsection 3.1.6). A central repository containing annotations for both the standard library and external packages aligns well with this approach. Inspiration can be drawn from Python's Typeshed (see Subsection 3.1.6) and Ruby's RBI Central (see Subsection 3.2.3). Such a repository could be hosted on a platform like GitHub, permitting contributions from the wider community.

4.4.2 Syntax

For prototyping purposes, adapting comments or special methods (like in Sorbet) might be the right approach to implementing gradual typing, due to its ease of implementation. However, for wide-scale adoption, inline annotations are preferable because of their conciseness, readability, and relative distance to the code.

All three studied type systems provide a universal supertype, which is helpful when annotating code bases and therefore worth considering for inclusion. Nullable types, NA-able (as proposed by Turcotte et al. [8]), and intersection types are features that allow for more granular typing. Given their relative rarity [8], non-nullable should be the default option with extra syntax to

mark their nullability. Intersection types are not explicitly mentioned by Turcotte et al. [8] but are featured in all 3 reviewed type systems and thus are potential candidates for inclusion.

Type aliases, union types, and type inference have several aspects in common: they lead to more concise code and are employed by Python, Sorbet, and PHP. This suggests that R should consider incorporating these features to improve conciseness.

4.4.3 Semantics

Two out of the three studied type systems can be considered type-safe, preventing type errors. Type soundness, on the other hand, is only observed to a limited extent among the three programming languages. Sorbet puts the most effort in this direction by offering a combination of static type checking and runtime checks. While it prevents array indexing errors [182], it does not prevent other classes of type errors, such as refinement invalidation errors [182]. Thus, it can only be considered sound using a very loose definition of the term [14].

There is always a trade-off between type soundness on one side and completeness and runtime performance on the other [182]. Developing both sound and unsound type systems are valid approach. A reasonable compromise might be to follow the lead of Sorbet, allowing programmers to adjust the strictness of type checks according to their needs. In this approach, for less strict settings soundness is optional, while runtime performance is imperative. For the strictest setting, soundness is mandatory (or at least an aspirational goal). This implies the hybrid type-enforcement strategy.

With the R language supporting polymorphism, the inclusion of both parametric and subtyping polymorphism into a gradual type system should be seriously considered. Especially as they are universally adopted across the studied type systems. Combining nominal and structural subtyping is also advisable as nominal subtyping is used by all three systems and structural subtyping by two-thirds of them. Structural subtyping using some form of Sorbet-style interfaces might be seen as an alternative for duck typing.

Type bounds, given their popularity across the studied type systems, should be also considered for inclusion. Variance and method overloading, however, have low adoption and should be considered secondary priorities. Although they might be good to eventually add, they should not be the starting point.

In the R language, complex compound types, such as matrices and data frames, are common [8]. Considering their popularity and special behaviors, it might be worth introducing them as stand-alone types in a gradual type system. Static annotations discussed in Subsection 3.1.5 (Pandas and Statically Typed Pandas packages), might serve as inspiration for handling such type annotations.

4.4.4 Adoption

Adding static types to a dynamically typed language may be met with cautiousness or even resistance from both users and developers of the language, especially if the community consists mainly of statisticians rather than computer scientists. After all, a similar reaction was observed in Python (see Subsection 3.1.2).

Several strategies employed by the languages discussed in Chapter 3 may help mitigate such sentiments. For instance, industry tools like Sorbet (Subsection 3.2.3) or Hack (Subsection 3.3.3) offered runtime performance optimizations and support for popular IDEs from the get-go. Hack even implemented changes unrelated to gradual typing (Subsection 3.3.3), such as fixing inconsistent behavior for trailing commas and introducing concise syntax for lambda expressions. Such improvements can make the adoption of gradual typing more palatable, even for programmers who are not yet fully convinced. If gradual typing is added to R, these strategies should be considered.

Another important factor is communication. Early discussions in the Python community (see Subsection 3.1.2) demonstrate the need to explain what the change entails for users' programming language and what it does not (e.g., types are and will remain optional), explain the benefits of the change, address concerns, and listen and react to feedback.

Two possible approaches to consider for R are the big bang approach and the evolution approach. With the big bang approach, a fully-featured type system is introduced (like the industry tools). With the evolution approach, features are gradually introduced to the language.

The big bang approach offers advantages such as users receiving competitive functionality from day one and potentially fewer compromises. However, it may take a long time to ship and miss regular feedback from the community. On the other hand, the evolution approach presents less drastic changes, mitigating knee-jerk reactions and providing continuous feedback from the community, but the initial product may be inferior in many aspects.

For R, given its governance model is closer to Python than to Sorbet or Hack, the evolution approach might make more sense. One possible way to implement this approach, inspired by Python, could be to start with syntax only, so that programmers may use it even if just for documentation purposes. Then, develop basic semantics alongside the emergence of type checkers, beginning with function parameters and return values, followed by variable declarations, and finally addressing classes, higher-order functions, and complex data types.

4.5 Summary

In conclusion, this section introduces the R programming language for statistical programming, its history, use cases, types, and the three object-oriented systems. Furthermore, it discusses the benefits of adding gradual typing to this language, such as improved reliability, developer experience, documentation, and potential performance optimizations.

However, implementing gradual typing in R brings several challenges, including complex data types like Data Frames and Matrices, three independent object-oriented systems each with different behavior that can be combined, and functions with many potentially polymorphic arguments. For annotating all of these constructs, a sweet spot between conciseness, comprehensibility for language users, and usefulness of the provided type information needs to be found.

A brief review of prior work is presented, focusing on Turcotte et al. [8], who not only introduced the first draft of a gradual type system for R but also tested it on a significant number of popular packages.

Lastly, this section discusses ideas for the next evolution of this concept. For the implementation strategy, this thesis advocates extending rather than exploiting R's syntax, utilizing a combination of compile-time and runtime checks, and offering optional strictness of the checks. Regarding syntax, this thesis supports the use of in-line annotations. For semantics, the recommendations include incorporating parametric polymorphism, nominal subtyping for regular object-oriented systems, structural subtyping using interfaces, and type bounds, and considering complex container types as stand-alone types.

By taking these recommendations into account, future work on gradual typing in R can build on the foundation laid by previous research and enhance the language with improved type safety, readability, and performance.

Chapter 5
Conclusion

This section summarizes all research conducted throughout this thesis. It summarizes findings and hints at opportunities for further research in the area.

R is a popular dynamically typed programming language for statistics. As such, it is used to inform critical decisions in areas such as policy, finance, public health, and healthcare. Having an error in one of its popular statistical packages could lead to a cornucopia of poor consequential decisions in all of these areas. A reliable static type system could prevent such issues while providing a better developer experience, improved documentation, and potential performance optimizations.

However, retrofitting a static type system to a dynamically typed programming language is a significant undertaking. In this thesis, cases were introduced where this process took many years or even decades. R's unique attributes, such as complex data types, three independent object-oriented systems, and a common practice of having more than 20 parameters per function, don't make this task any easier. Success involves selecting the right implementation strategy, adjusting or working around the language's syntax and semantics, and supporting adoption through a compelling value proposition for programmers and transparent communication with the language community.

Implementation strategy decisions include whether to add types to the existing language or create a new separate typed language, whether to integrate static type checking into the language or provide it as a separate tool, and whether to perform type checks during runtime, compile time, or both. Syntax considerations encompass exploiting existing syntax through comments, methods, or decorators, or expanding it with a special syntax for type annotations. Deciding whether or not to employ a type inference algorithm and evaluating the use of features such as union types, intersection types, and type aliasing are also important aspects of syntax design. Semantic considerations involve determining rules for type compatibility, establishing rules for type conversions and type coercions, and annotating complex composite types both accurately and concisely. Selecting a type-enforcement strategy for interactions between statically and dynamically typed code and considering whether type-soundness should be a goal and, if so, how to ensure it, are also critical aspects of designing a gradual typing system for R.

In Python's history, the gradual typing effort was initiated by the language creator, starting with discussions within the community. Subsequently, the syntax for typing was introduced, and the first static type checkers emerged. Semantics were added to the core language only later, with mypy becoming the reference type checker. The Python type system has continuously evolved, with typing features being introduced in every new major release. Python's approach involves enhancing the language gradually and leaving static type checking to external tools. Language semantics and type annotations for the standard library as well as for external packages are

managed in a central repository.

The history of gradual typing in Ruby began in academia with Diamondback Ruby shortly followed by several other experimental tools. The commercial sphere, exemplified by Sorbet, built on the work done by academia, adapting ideas and type definitions while prioritizing runtime performance. The core language team joined the effort later, introducing a separate language for type annotations - RBS. Sorbet's approach involved sharing a mature product with the community, exploiting existing syntax with special methods, integrating static and dynamic checking, offering optional strictness of type checks, and centrally managing standard definitions. Ruby's approach involved keeping annotations outside the language itself, introducing a special type annotation language (RBS), and leaving type checking to external tools.

PHP's gradual typing history started in the commercial sphere with Facebook's Hack language, focusing on runtime speed and IDE support. The core language responded by introducing its own spin on gradual typing features. Similarly to Python, they are evolving their gradual type system gradually. In parallel, static type checkers like PHPStan emerged. Gradual typing in PHP is continuing its evolution, with new features introduced in every major version. PHP's approach involves starting small (method annotations and basic types) and gradually enhancing the type system. The core language performs runtime type checks and leaves static checks to external tools. Similarly to Sorbet, it offers optional strictness of type checks.

While each of the gradual type systems took distinct approaches and implementation strategies, some decisions are common to all of them. In terms of syntactic features, all three languages support statically typed method parameters and return values, a universal supertype, nullable types, type aliasing, union types, intersection types, and type inference. In terms of semantic features, all languages strive for type safety and offer parametric polymorphism, nominal subtyping, and type bounds.

Based on the insights gleaned from other languages and prior research, this thesis suggests several key takeaways for implementing gradual typing in R. In terms of the implementation strategy, extending the existing syntax for conciseness and readability could be advantageous in winning over the existing user base. A combination of compile-time and runtime type checks, alongside an optional level of strictness, may offer an optimal balance between safety guarantees and runtime performance. Furthermore, managing type definitions and semantics centrally while providing type checking outside the language could align well with R's free software philosophy.

Regarding syntax, adapting comments or special methods might be suitable for initial prototyping purposes, while inline annotations could be a better option for wide-scale adoption. A special syntax for marking types like Nullable and NA-able types should be given serious consideration. Union types, intersection types, type aliases, and type inference should also be considered.

With respect to semantics, the study proposes embracing polymorphism and subtyping, as well as utilizing structural subtyping for interfaces to serve as a static alternative to duck-typing. The incorporation of type bounds is also recommended.

For adoption, several strategies can be employed to mitigate resistance from the existing community. These include introducing gradual typing alongside other improvements, supporting popular IDEs, and maintaining open two-way communication with the language community.

Potential areas for further research encompass improving the type system proposed by Turcotte et al. [8], focusing on further exploring compatibility, conversion, and coercion rules as well as appropriate syntax and semantics for annotating matrix and data frame types (Subsection 3.1.5 could provide inspiration).

In summary, the author believes that by presenting the three case studies and offering these suggestions, he contributed, to the overarching goal of adding gradual typing to the R programming language.

Bibliography

1. CHENG, Javran. Adding Gradual Types to Existing Languages. 2018. Available also from: https://www.cs.umd.edu/sites/default/files/scholarly_papers/Cheng,%20Javran_1801.pdf. Accessed on March 26th, 2023.
2. SWAMY, Nikhil; FOURNET, Cedric; RASTOGI, Aseem; BHARGAVAN, Karthikeyan; CHEN, Juan; STRUB, Pierre-Yves; BIERMAN, Gavin. Gradual typing embedded securely in JavaScript. In: ACM New York, NY, USA, 2014, vol. 49, pp. 425–437. No. 1.
3. VITOUSEK, Michael M; KENT, Andrew M; SIEK, Jeremy G; BAKER, Jim. Design and evaluation of gradual typing for Python. In: *Proceedings of the 10th ACM Symposium on Dynamic languages*. 2014, pp. 45–56.
4. BONNAIRE-SERGEANT, Ambrose; DAVIES, Rowan; TOBIN-HOCHSTADT, Sam. Practical Optional Types for Clojure. In: *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 68–94. ISBN 9783662494974.
5. GIORGI, Federico M; CERAOLO, Carmine; MERCATELLI, Daniele. The R language: an engine for bioinformatics and data science. *Life*. 2022, vol. 12, no. 5, p. 648.
6. MORANDAT, Floréal; HILL, Brandon; OSVALD, Leo; VITEK, Jan. Evaluating the design of the R language: Objects and functions for data analysis. In: *ECOOP 2012–Object-Oriented Programming: 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings 26*. Springer, 2012, pp. 104–131.
7. TURCOTTE, Alexi; VITEK, Jan. Towards a Type System for R. In: 2019, pp. 1–5. ISBN 978-1-4503-6862-9. Available from DOI: 10.1145/3340670.3342426.
8. TURCOTTE, Alexi; GOEL, Aviral; KŘIKAVA, Filip; VITEK, Jan. Designing types for R, empirically. *Proceedings of the ACM on Programming Languages*. 2020, vol. 4, no. OOPSLA, pp. 1–25.
9. GITHUB, Inc. *Top languages used in 2022*. [N.d.]. Available also from: https://octoverse.github.com/2022/top-programming-languages. Accessed on April 15th, 2023.
10. INC, Stack Exchange. *2022 Developer Survey*. [N.d.]. Available also from: https://survey.stackoverflow.co/2022/#most-popular-technologies-language. Accessed on April 15th, 2023.
11. BV, TIOBE Software. *TIOBE Index for April 2023*. [N.d.]. Available also from: https://www.tiobe.com/tiobe-index/. Accessed on April 15th, 2023.
12. SCOTT, Michael L. *Programming Language Pragmatics*. 4th. Elsevier, 2015.
13. MILNER, Robin. A theory of type polymorphism in programming. *Journal of computer and system sciences*. 1978, vol. 17, no. 3, pp. 348–375.

14. KRISHNAMURTHI, Shriram; LERNER, Benjamin S.; POLITZ, Joe Gibbs. *Safety and Soundness*. 2020. Available also from: `https://papl.cs.brown.edu/2020/safety-soundness.html`. Accessed on April 22, 2023.

15. KRISHNAMURTHI, Shriram; LERNER, Benjamin S.; POLITZ, Joe Gibbs. *From Repeated Expressions to Functions*. 2020. Available also from: `https://papl.cs.brown.edu/2020/From_Repeated_Expressions_to_Functions.html`. Accessed on April 21, 2023.

16. UNIVERSITY, Northeastern. *Lecture Notes: Type Inference* [Course Notes]. 2019. Available also from: `https://course.ccs.neu.edu/cs4410sp19/lec_type-inference_notes.html`. Accessed on May 5th, 2023.

17. DAMAS, Luis; MILNER, Robin. Principal Type-Schemes for Functional Programs. In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Albuquerque, New Mexico: Association for Computing Machinery, 1982, pp. 207–212. POPL '82. ISBN 0897910656. Available from DOI: `10.1145/582153.582176`.

18. FLOMEBUL; KIPP, Chris; LACHKAR, Meriam; ZIJST, Erik van; HAWLEY, Aaron S.; WIJNAND, Dale; FOSTER, Ryan; BAALMAN, Philippus; HADŽIAVDIĆ, Sakib; MILLER, Heather. *Upper Type Bounds*. Scala Documentation, [n.d.]. Available also from: `https://docs.scala-lang.org/tour/upper-type-bounds.html`. Accessed on May 10, 2023.

19. PIERCE, Benjamin C. *Types and programming languages*. MIT press, 2002.

20. SIEK, Jeremy; TAHA, Walid. Gradual typing for functional languages. In: 2006.

21. TOBIN-HOCHSTADT, Sam; FELLEISEN, Matthias. The design and implementation of typed scheme. In: 2008, vol. 43, pp. 395–406. Available from DOI: `10.1145/1328897.1328486`.

22. ORTIN, Francisco; ZAPICO, Daniel; GARCÍA PEREZ-SCHOFIELD, Baltasar; RODRÍGUEZ, Miguel. Including both Static and Dynamic Typing in the same Programming Language. *Software, IET*. 2010, vol. 4, pp. 268–282. Available from DOI: `10.1049/iet-sen.2009.0070`.

23. TAKIKAWA, Asumu; FELTEY, Daniel; GREENMAN, Ben; NEW, Max S; VITEK, Jan; FELLEISEN, Matthias. Is sound gradual typing dead? In: *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*. 2016, pp. 456–468.

24. GREENMAN, Ben; DIMOULAS, Christos; FELLEISEN, Matthias. Typed–Untyped Interactions: A Comparative Analysis. *ACM Transactions on Programming Languages and Systems*. 2023, vol. 45, no. 1, pp. 1–54.

25. PYTHON.ORG. *Python Applications*. 2023. Available also from: `https://www.python.org/about/apps/`. Accessed on April 2nd, 2023.

26. ROSSUM, Guido van; TEAM, Python Development. *Python tutorial*. Python Software Foundation, 2023. Version 3.11.3.

27. ROSSUM, Guido van. *A Brief Timeline of Python*. [N.d.]. Available also from: `https://python-history.blogspot.com/2009/01/brief-timeline-of-python.html`. Accessed on April 15th, 2023.

28. ROSSUM, Guido van; TEAM, Python Development. *The Python Language Reference*. Python Software Foundation, 2023. Version 3.10.11.

29. ROSSUM, Guido van; TEAM, Python Development. *The Python Library Reference*. Python Software Foundation, 2023. Version 3.11.3.

30. ROSSUM, Guido van. Adding Optional Static Typing to Python. *Artima Weblogs*. 2004. Available also from: https://www.artima.com/weblogs/viewpost.jsp?thread=85551. Accessed on March 26th, 2023.

31. ARTIMA. *Forum: Adding Optional Static Typing to Python*. [N.d.]. Available also from: https://www.artima.com/forums/flat.jsp?forum=106&thread=85551. Accessed on April 1st, 2023.

32. ROSSUM, Guido van. Adding Optional Static Typing to Python – Part II. *Artima Weblogs*. 2005. Available also from: https://www.artima.com/weblogs/viewpost.jsp?thread=86641. Accessed on March 26th, 2023.

33. ARTIMA. *Forum: Adding Optional Static Typing to Python – Part II*. [N.d.]. Available also from: https://www.artima.com/forums/flat.jsp?forum=106&thread=87182&start=0&msRange=15. Accessed on April 1st, 2023.

34. ROSSUM, Guido van. Optional Static Typing – Stop the Flames! *Artima Weblogs*. 2005. Available also from: https://www.artima.com/weblogs/viewpost.jsp?thread=87182. Accessed on March 26th, 2023.

35. ARTIMA. *Forum: Optional Static Typing – Stop the Flames!* [N.d.]. Available also from: https://www.artima.com/forums/flat.jsp?forum=106&thread=86641&start=45&msRange=15. Accessed on April 1st, 2023.

36. PYTHON.ORG. *Index of Python Enhancement Proposals (PEPs)* [Python Enhancement Proposal]. 2000. Available also from: https://peps.python.org/pep-0000/#introduction. Accessed on April 2nd, 2023.

37. WINTER, Collin. *pypi/typecheck - version history*. [N.d.]. Available also from: https://pypi.org/project/typecheck/#history. Accessed on April 16th, 2023.

38. WINTER, Collin. *Internet Archive - Type-checking module for Python*. [N.d.]. Available also from: http://web.archive.org/web/20070730120117/http://oakwinter.com/code/typecheck/. Accessed on April 16th, 2023.

39. BIRCH, Bill. *We need structural subtyping NOW!* [N.d.]. Available also from: https://billbirch.wordpress.com/2006/03/09/we-need-structural-subtyping-now/. Accessed on April 16th, 2023.

40. ROSSUM, Guido van. Python 3000 - Adaptation or Generic Functions? *Artima Weblogs*. 2006. Available also from: https://www.artima.com/weblogs/viewpost.jsp?thread=155123. Accessed on April 16th, 2023.

41. BIRCH, Bill. *Unsure if I should write a PEP on Types*. [N.d.]. Available also from: https://mail.python.org/pipermail/python-3000/2006-April/001328.html. Accessed on April 16th, 2023.

42. BIRCH, Bill. *Some Thoughts on Types for Python-3000*. [N.d.]. Available also from: https://billbirch.wordpress.com/2006/05/01/draft-pep-types-for-python-3000/. Accessed on April 16th, 2023.

43. WINTER, Collin. *Questions on optional type annotations*. [N.d.]. Available also from: https://mail.python.org/pipermail/python-3000/2006-May/001972.html. Accessed on April 16th, 2023.

44. WINTER, Collin. *Type annotations: annotating generators*. [N.d.]. Available also from: https://mail.python.org/pipermail/python-3000/2006-May/002091.html. Accessed on April 16th, 2023.

45. ROSSUM, Guido van. *Type annotations: annotating generators*. [N.d.]. Available also from: https://mail.python.org/pipermail/python-3000/2006-May/002103.html. Accessed on April 16th, 2023.

46. WINTER, Collin. *Mixing annotations and non-annotations*. [N.d.]. Available also from: https://mail.python.org/pipermail/python-3000/2006-May/002209.html. Accessed on April 16th, 2023.
47. WINTER, Collin. *Third-party annotation libraries vs the stdlib*. [N.d.]. Available also from: https://mail.python.org/pipermail/python-3000/2006-June/002438.html. Accessed on April 16th, 2023.
48. WINTER, Collin. *Type parameterization (was: Re: Type annotations: annotating generators)*. [N.d.]. Available also from: https://mail.python.org/pipermail/python-3000/2006-May/002105.html. Accessed on April 16th, 2023.
49. WINTER, Collin; LOWNDS, Tony. *Function Annotations* [Python Enhancement Proposal 3107]. 2006. Available also from: https://www.python.org/dev/peps/pep-3107/. Accessed on March 26th, 2023.
50. ROSSUM, Guido van; LEHTOSALO, Jukka; LANGA, Łukasz. *Type Hints* [Python Enhancement Proposal 484]. 2014. Available also from: https://www.python.org/dev/peps/pep-0484/. Accessed on March 26th, 2023.
51. ROSSUM, Guido van; LEVKIVSKYI, Ivan. *The Theory of Type Hints* [Python Enhancement Proposal 483]. 2014. Available also from: https://www.python.org/dev/peps/pep-0483/. Accessed on March 26th, 2023.
52. LANGA, Łukasz. *Literature Overview for Type Hints* [Python Enhancement Proposal 482]. 2015. Available also from: https://peps.python.org/pep-0482/. Accessed on April 7th, 2023.
53. GONZALEZ, Ryan; HOUSE, Philip; LEVKIVSKYI, Ivan; ROACH, Lisa; ROSSUM, Guido van. *Syntax for Variable Annotations* [Python Enhancement Proposal 526]. 2016. Available also from: https://www.python.org/dev/peps/pep-0526/. Accessed on March 26th, 2023.
54. LEVKIVSKYI, Ivan. *Core support for typing module and generic types* [Python Enhancement Proposal 560]. 2017. Available also from: https://www.python.org/dev/peps/pep-0560/. Accessed on April 8th, 2023.
55. LANGA, Łukasz. *Postponed Evaluation of Annotations* [Python Enhancement Proposal 563]. 2017. Available also from: https://www.python.org/dev/peps/pep-0563/. Accessed on April 8th, 2023.
56. SMITH, Ethan. *Distributing and Packaging Type Information* [Python Enhancement Proposal 561]. 2017. Available also from: https://peps.python.org/pep-0561/. Accessed on April 28th, 2023.
57. LEVKIVSKYI, Ivan; LEHTOSALO, Jukka; LANGA, Łukasz. *Type Hinting Generics In Standard Collections* [Python Enhancement Proposal 544]. 2017. Available also from: https://www.python.org/dev/peps/pep-0544/. Accessed on April 8th, 2023.
58. LEHTOSALO, Jukka. *TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys* [Python Enhancement Proposal 589]. 2019. Available also from: https://www.python.org/dev/peps/pep-0589/. Accessed on April 8th, 2023.
59. LANGA, Łukasz. *Type Hinting Generics In Standard Collections* [Python Enhancement Proposal 585]. 2019. Available also from: https://www.python.org/dev/peps/pep-0585/. Accessed on April 8th, 2023.
60. PRADOS, Philippe; MOSS, Maggie. *Allow writing union types as X — Y* [Python Enhancement Proposal 604]. 2019. Available also from: https://www.python.org/dev/peps/pep-0604/. Accessed on April 8th, 2023.

61. MENDOZA, Mark. *Parameter Specification Variables* [Python Enhancement Proposal 612]. 2019. Available also from: https://www.python.org/dev/peps/pep-0612/. Accessed on April 8th, 2023.

62. ZHU, Shannon. *Explicit Type Aliases* [Python Enhancement Proposal 613]. 2020. Available also from: https://www.python.org/dev/peps/pep-0613/. Accessed on April 8th, 2023.

63. MENDOZA, Mark; RAHTZ, Matthew; SRINIVASAN, Pradeep Kumar; SILES, Vincent. *Variadic Generics* [Python Enhancement Proposal 646]. 2020. Available also from: https://www.python.org/dev/peps/pep-0646/. Accessed on April 8th, 2023.

64. FOSTER, David. *Marking individual TypedDict items as required or potentially-missing* [Python Enhancement Proposal 655]. 2021. Available also from: https://www.python.org/dev/peps/pep-0655/. Accessed on April 8th, 2023.

65. SRINIVASAN, Pradeep Kumar; HILTON-BALFE, James. *Self Type* [Python Enhancement Proposal 673]. 2021. Available also from: https://www.python.org/dev/peps/pep-0673/. Accessed on April 8th, 2023.

66. DI GRAZIA, Luca; PRADEL, Michael. The evolution of type annotations in python: an empirical study. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 209–220.

67. TEAM, The pandas development. *pandas: Flexible and powerful data analysis / manipulation library for Python*. 2021. Available also from: https://pandas.pydata.org. Accessed on May 6th, 2023.

68. DEVELOPERS, NumPy. *NumPy*. 2021. Available also from: https://numpy.org. Accessed on May 6th, 2023.

69. KERZOL, Mate. *Strictly Typed Pandas*. 2021. Available also from: https://strictly-typed-pandas.readthedocs.io/en/latest/index.html. Accessed on May 6th, 2023.

70. CONTRIBUTORS, Typeshed. *Typeshed*. 2023. Available also from: https://github.com/python/typeshed. Accessed on April 28th, 2023.

71. CONTRIBUTORS, Typeshed. *Contributing to Typeshed*. 2021. Available also from: https://github.com/python/typeshed/blob/main/CONTRIBUTING.md. Accessed on April 28th, 2023.

72. PROJECT, the mypy. *mypy official website*. 2023. Available also from: https://www.mypy-lang.org/index.html. Accessed on April 9th, 2023.

73. LEHTOSALO, Jukka. *mypy documentation*. 2022. Available also from: https://mypy.readthedocs.io/en/latest/#. Accessed on April 16th, 2023.

74. GOOGLE. *pypi/pytype - version history*. [N.d.]. Available also from: https://pypi.org/project/pytype/#history. Accessed on April 9th, 2023.

75. GOOGLE. *pytype official website*. [N.d.]. Available also from: https://google.github.io/pytype/. Accessed on April 9th, 2023.

76. MICROSOFT. *pypi/pyright - version history*. [N.d.]. Available also from: https://pypi.org/project/pyright/#history. Accessed on April 9th, 2023.

77. TRAUT, Eric; VOLODIN, Dmitry; ZIJLSTRA, Zelle; CLAUSS, Christian. *Differences Between Pyright and Mypy*. 2023. Available also from: https://github.com/microsoft/pyright/blob/main/docs/mypy-comparison.md. Accessed on April 9th, 2023.

78. SOURCE, Meta Open. *Pyre official website*. [N.d.]. Available also from: https://pyre-check.org/. Accessed on April 9th, 2023.

79. SOURCE, Meta Open. *Pyre Documentation - Types in Python*. [N.d.]. Available also from: https://pyre-check.org/docs/types-in-python/. Accessed on April 9th, 2023.

80. SOURCE, Meta Open. *pypi/pyre - version history*. [N.d.]. Available also from: `https://pypi.org/project/pyre-check/#history`. Accessed on April 9th, 2023.
81. RAK-AMNOUYKIT, Ingkarat; MCCREVAN, Daniel; MILANOVA, Ana; HIRZEL, Martin; DOLBY, Julian. Python 3 Types in the Wild: A Tale of Two Type Systems. In: *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*. Virtual, USA: Association for Computing Machinery, 2020, pp. 57–70. DLS 2020. ISBN 9781450381758. Available from DOI: `10.1145/3426422.3426981`.
82. MATSUMOTO, Yukihiro; COMMUNITY, The Ruby. *About Ruby*. 2023. Available also from: `https://www.ruby-lang.org/en/about/`. Accessed on April 23th, 2023.
83. FLANAGAN, David; MATSUMOTO, Yukihiro. *The Ruby Programming Language: Everything You Need to Know*. " O'Reilly Media, Inc.", 2008.
84. EDGAR, Michael Joseph. *Static Analysis for Ruby in the Presence of Gradual Typing*. 2011. Available also from: `https://digitalcommons.dartmouth.edu/cgi/viewcontent.cgi?article=1071&context=senior_theses`. Undergraduate Theses. Dartmouth College.
85. SIEGER, Nick. *RubyConf: History of Ruby*. 2006. Available also from: `http://blog.nicksieger.com/articles/2006/10/20/rubyconf-history-of-ruby/`. Accessed on April 26th.
86. ENDOH, Yusuke. *About Ruby*. 2013. Available also from: `https://www.ruby-lang.org/en/news/2013/02/24/ruby-2-0-0-p0-is-released/`. Accessed on April 23th, 2023.
87. NARUSE. *About Ruby*. 2020. Available also from: `https://www.ruby-lang.org/en/news/2020/12/25/ruby-3-0-0-released/`. Accessed on April 23th, 2023.
88. HANSSON, David Heinemeier; RAILS COMMUNITY, Ruby on. *About Ruby*. 2023. Available also from: `https://rubyonrails.org`. Accessed on April 23th, 2023.
89. TEAM, Ruby Core. *Integer - Ruby 3.2.2*. 2023. Available also from: `https://ruby-doc.org/3.2.2/Integer.html`. Accessed on April 28th.
90. TEAM, Ruby Core. *Float - Ruby 3.2.2*. 2023. Available also from: `https://ruby-doc.org/3.2.2/Float.html`. Accessed on April 28th.
91. TEAM, Ruby Core. *Complex - Ruby 3.2.2*. 2023. Available also from: `https://ruby-doc.org/3.2.2/Complex.html`. Accessed on April 28th.
92. TEAM, Ruby Core. *Rational - Ruby 3.2.2*. 2023. Available also from: `https://ruby-doc.org/3.2.2/Rational.html`. Accessed on April 28th.
93. FURR, Michael; AN, Jong-hoon (David); FOSTER, Jeffrey S.; HICKS, Michael. Static Type Inference for Ruby. In: *Proceedings of the 2009 ACM Symposium on Applied Computing*. Honolulu, Hawaii: Association for Computing Machinery, 2009, pp. 1859–1866. SAC '09. ISBN 9781605581668. Available from DOI: `10.1145/1529282.1529700`.
94. AN, Jong-hoon; CHAUDHURI, Avik; FOSTER, Jeffrey S. Static Typing for Ruby on Rails. In: *2009 IEEE/ACM International Conference on Automated Software Engineering*. 2009, pp. 590–594. Available from DOI: `10.1109/ASE.2009.80`.
95. FURR, Michael; AN, Jong-hoon (David); DALY, Mark; KIRZHNER, Benjamin; CHAUDHURI, Avik; FOSTER, Jeffrey S.; HICKS, Michael. *Diamondback Ruby official website*. [N.d.]. Available also from: `http://www.cs.umd.edu/projects/PL/druby/`. Accessed on April 12th, 2023.
96. HECHT, Matthew S. *Flow analysis of computer programs*. Elsevier Science Inc., 1977.
97. CYTRON, Ron; FERRANTE, Jeanne; ROSEN, Barry K; WEGMAN, Mark N; ZADECK, F Kenneth. An efficient method of computing static single assignment form. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, pp. 25–35.

98. EDGAR, Michael Joseph. *LASER: Static Type Analysis Tool for Ruby*. 2021. Available also from: https://github.com/michaeledgar/laser. Accessed on April 23th, 2023.

99. AN, Jong-hoon (David); CHAUDHURI, Avik; FOSTER, Jeffrey S.; HICKS, Michael. Dynamic Inference of Static Types for Ruby. In: Austin, Texas, USA: Association for Computing Machinery, 2011, pp. 459–472. POPL '11. ISBN 9781450304900. Available from DOI: 10.1145/1926385.1926437.

100. REN, Brianna M.; TOMAN, John; STRICKLAND, Stephen T.; FOSTER, Jeffrey S. The ruby type checker. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. 2013, pp. 1565–1572.

101. STRICKLAND, Tess; FOSTER, Jeffrey S.; JTERRY64. *The Ruby Type Checker* [https://github.com/plum-umd/rtc]. GitHub, 2023. Accessed on April 29th, 2023.

102. FOSTER, Jeffrey S.; RDL CONTRIBUTORS, the. *RDL: Ruby Dynamic Types*. RubyGems, 2023. Available also from: https://rubygems.org/gems/rdl/versions. Accessed on April 29th, 2023.

103. FOSTER, Jeffrey S.; REN, Brianna M.; STRICKLAND, Stephen T.; YU, Alexander T.; KAZEROUNIAN, Milod; GURIA, Sankha Narayan. *RDL: A Type System for Ruby* [https://github.com/tupl-tufts/rdl]. GitHub, 2023.

104. KAZEROUNIAN, Milod; GURIA, Sankha Narayan; VAZOU, Niki; FOSTER, Jeffrey S; VAN HORN, David. Type-level computations for Ruby libraries. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 966–979.

105. PETRASHKO, Dmitry; TARJAN, Paul. *Gradual Typing of Ruby at Scale*. 2018. Available also from: https://www.thestrangeloop.com/2018/gradual-typing-of-ruby-at-scale.html. Accessed on April 29th, 2023.

106. GURIA, Sankha; KAZEROUNIAN, Milod; VAN HORN, David; FOSTER, Jeffrey S. *Ruby Dynamic Language*. [N.d.]. Available also from: https://plum-umd.github.io/projects/rdl.html. Accessed on April 29th, 2023.

107. ZIMMERMAN, Jake. *Sorbet: Stripe's type checker for Ruby*. 2019. Available also from: https://stripe.com/blog/sorbet-stripes-type-checker-for-ruby. Accessed on April 29th, 2023.

108. PETRASHKO, Dmitry; TARJAN, Paul; ELHAGE, Nelson. *A Practical Type System for Ruby at Stripe*. 2018. Available also from: https://rubykaigi.org/2018/presentations/DarkDimius.html#may31. Accessed on April 30th, 2023.

109. IRY, James. *State of Sorbet: Spring 2019*. 2019. Available also from: https://sorbet.org/blog/2019/05/16/state-of-sorbet-spring-2019. Accessed on April 29th, 2023.

110. STRIPE. *sorbet*. RubyGems, 2023. Available also from: https://rubygems.org/gems/sorbet. Accessed on April 29th, 2023.

111. ZIMMERMAN, Jake; TARJAN, Paul. *State of Sorbet: A Type Checker for Ruby*. 2019. Available also from: https://rubykaigi.org/2019/presentations/jez.html#apr19. Accessed on April 30th, 2023.

112. SHEA, Connor; CONTRIBUTORS, sorbet-typed. *Sorbet-typed*. GitHub, 2022. Available also from: https://github.com/sorbet/sorbet-typed. Accessed on April 29th, 2023.

113. SORBET. *RBI Files*. Sorbet, 2023. Available also from: https://sorbet.org/docs/rbi#hand-written-rbis-for-gems. Accessed on April 30th, 2023.

114. SHOPIFY. *Tapioca*. 2023. Available also from: https://github.com/Shopify/tapioca. Accessed on April 30th, 2023.

115. PETRASHKO, Dmitry. *Fast typechecking for Ruby*. 2019. Available also from: https://sorbet.org/docs/talks/jvm-ls-2019. Accessed on April 30th, 2023.

116. SOUTARO, Matsumoto. *Steep: Gradual Typing for Ruby*. 2023. Available also from: https://github.com/soutaro/steep. Accessed on May 1st, 2023.

117. KOHL, Michael. *Ruby Magic: Type Checking in Ruby*. 2019. Available also from: https://blog.appsignal.com/2019/08/27/ruby-magic-type-checking-in-ruby.html. Accessed on May 1st, 2023.

118. MATSUMOTO, Soutaro. *The State of Ruby 3 Typing*. 2020. Available also from: https://developer.squareup.com/blog/the-state-of-ruby-3-typing/. Accessed on April 30th, 2023.

119. ZIMMERMAN, Jake. *Ruby 3, RBS, and Sorbet*. 2020. Available also from: https://sorbet.org/blog/2020/07/30/ruby-3-rbs-sorbet/. Accessed on May 1st, 2023.

120. RUBY-LANG.ORG. *Ruby 3.0.0 Released*. 2020. Available also from: https://www.ruby-lang.org/en/news/2020/12/25/ruby-3-0-0-released/. Accessed on May 1st, 2023.

121. SORBET. *Open Sourcing the Sorbet Compiler*. 2021. Available also from: https://sorbet.org/blog/2021/07/30/open-sourcing-sorbet-compiler. Accessed on May 1st, 2023.

122. RUBY-LANG.ORG. *Ruby 3.1.0 Released*. 2021. Available also from: https://www.ruby-lang.org/en/news/2021/12/25/ruby-3-1-0-released/. Accessed on May 1st, 2023.

123. SORBET. *Open Sourcing Sorbet for VSCode*. 2022. Available also from: https://sorbet.org/blog/2022/01/06/open-sourcing-sorbet-vscode. Accessed on May 1st, 2023.

124. ZIMMERMAN, Jake. *Tapioca is the recommended way to generate RBIs for Sorbet*. 2022. Available also from: https://sorbet.org/blog/2022/07/27/srb-tapioca. Accessed on May 1st, 2023.

125. SOUTARO, Matsumoto. *steep*. RubyGems, 2023. Available also from: https://rubygems.org/gems/steep/. Accessed on May 1st, 2023.

126. STRIPE. *Sorbet - Official Website*. 2023. Available also from: https://sorbet.org. Accessed on May 1st, 2023.

127. BLANCA, Rafael; SHOPIFY. *tapioca*. RubyGems, 2023. Available also from: https://rubygems.org/gems/tapioca. Accessed on May 1st, 2023.

128. WEISSMANN, Ben. *Typing the Untyped: Soundness in Gradual Type Systems*. 2019. Available also from: https://www.thestrangeloop.com/2019/typing-the-untyped-soundness-in-gradual-type-systems.html. Accessed: yyyy-mm-dd.

129. LERDORF, Rasmus; TATROE, Kevin; KAEHMS, Bob; MCGREDY, Ric. *Programming Php*. " O'Reilly Media, Inc.", 2002.

130. ZEND. *About Zend by Perforce*. 2023. Available also from: https://www.zend.com/about. Accessed on May 3rd, 2023.

131. GROUP, PHP. *PHP: Supported Versions*. 2021. Available also from: https://prototype.php.net/versions/. Accessed on May 3rd, 2023.

132. FOUNDATION, The PHP. *The PHP Foundation*. 2021. Available also from: https://thephp.foundation/. Accessed on May 3rd, 2023.

133. *PHP: Language Reference - Types*. The PHP Group, 2023. Available also from: https://www.php.net/manual/en/language.types.intro.php. Accessed on May 3rd, 2023.

134. *PHP: Language Reference - Type System*. The PHP Group, 2023. Available also from: https://www.php.net/manual/en/language.types.type-system.php. Accessed on May 3rd, 2023.

135. GROUP, The PHP. *PHP: Arrays - Manual*. 2023. Available also from: https://www.php.net/manual/en/language.types.array.php. Accessed on May 3rd, 2023.

136. GROUP, The PHP. *PHP: Strings - Manual*. 2023. Available also from: https://www.php.net/manual/en/language.types.string.php. Accessed on May 3rd, 2023.

137. GROUP, The PHP. *PHP: Numeric strings - Manual*. 2023. Available also from: https://www.php.net/manual/en/language.types.numeric-strings.php. Accessed on May 3rd, 2023.

138. EVANS, Jason. *The HipHop Virtual Machine* [Blog]. 2011. Available also from: https://engineering.fb.com/2011/12/09/open-source/the-hiphop-virtual-machine/. Accessed on May 3rd, 2023.

139. VERLAUGET, Julien; MENGHRAJANI, Alok. *Hack: A new programming language for HHVM*. 2014. Available also from: https://engineering.fb.com/2014/03/20/developer-tools/hack-a-new-programming-language-for-hhvm/. Accessed on May 3rd, 2023.

140. POBAR, Joel. *Hack Developer Day Recap* [Blog]. 2014. Available also from: https://engineering.fb.com/2014/04/10/developer-tools/hack-developer-day-recap/. Accessed on May 3rd, 2023.

141. IYENGAR, Vishnu. *Announcing the Hack Transpiler* [Blog]. 2014. Available also from: https://engineering.fb.com/2014/11/11/developer-tools/announcing-the-hack-transpiler/. Accessed on May 3rd, 2023.

142. GREEN, Erin. *HHVM Adoption News* [Blog]. 2015. Available also from: https://engineering.fb.com/2015/04/17/data-center-engineering/hhvm-adoption-news/. Accessed on May 3rd, 2023.

143. *PHP: PHP 7 ChangeLog*. [N.d.]. Available also from: https://www.php.net/ChangeLog-7.php#PHP_7_0. Accessed on May 5th, 2023.

144. TECHNOLOGIES, Zend. *PHP 7: What's New and How It Impacts Your Code* [https://www.zend.com/blog/php-7]. 2015.

145. PHP GROUP. *PHP: Type Declarations - Manual*. 2021. Available also from: https://www.php.net/manual/en/language.types.declarations.php. Accessed on May 5th, 2023.

146. MADURO, Nuno. *[10.x] Uses PHP Native Type Declarations*. GitHub, 2023. Available also from: https://github.com/laravel/laravel/pull/6010. GitHub repository, Accessed on May 6th, 2023.

147. OTWELL, Taylor; CONTRIBUTORS, Laravel. *Laravel*. GitHub, 2011. Available also from: https://github.com/laravel/laravel. Accessed on May 6th, 2023.

148. TSCHAN, Sebastian. *jQuery File Upload*. GitHub, 2021. Available also from: https://github.com/blueimp/jQuery-File-Upload. GitHub repository, Accessed on May 6th, 2023.

149. OTWELL, Taylor; CONTRIBUTORS, Laravel. *Laravel Framework*. GitHub, 2023. Available also from: https://github.com/laravel/framework. GitHub repository, Accessed on May 6th, 2023.

150. CONTRIBUTORS, Symfony. *Symfony*. GitHub, 2023. Available also from: https://github.com/symfony/symfony. Accessed on May 6th, 2023.

151. CONTRIBUTORS, Composer. *Composer*. GitHub, 2023. Available also from: https://github.com/composer/composer. Accessed on May 6th, 2023.

152. ZANINOTTO, Francois; CONTRIBUTORS, Faker. *Faker*. GitHub, 2020. Available also from: https://github.com/fzaninotto/Faker. Accessed on May 6th, 2023.

153. DOWLING, Michael; CONTRIBUTORS, Guzzle. *Guzzle*. GitHub, 2023. Available also from: https://github.com/guzzle/guzzle. Accessed on May 6th, 2023.

154. CONTRIBUTORS, Nextcloud. *Nextcloud Server*. GitHub, 2023. Available also from: https://github.com/nextcloud/server. Accessed on May 6th, 2023.

155. CONTRIBUTORS, DesignPatternsPHP. *DesignPatternsPHP*. GitHub, 2023. Available also from: https://github.com/DesignPatternsPHP/DesignPatternsPHP. Accessed on May 6th, 2023.

156. BOGGIANO, Jordi; CONTRIBUTORS, Monolog. *Monolog*. GitHub, 2023. Available also from: https://github.com/Seldaek/monolog. Accessed on May 6th, 2023.

157. *PHPStan - PHP Static Analysis Tool*. [N.d.]. Available also from: https://phpstan.org. Accessed on May 5th, 2023.

158. *Psalm - A static analysis tool for finding errors in PHP applications*. [N.d.]. Available also from: https://psalm.dev. Accessed on May 5th, 2023.

159. CONTRIBUTORS, Phan. *Phan - A static analyzer for PHP*. 2023. Available also from: https://github.com/phan/phan. Accessed on May 5th, 2023.

160. CONTRIBUTORS, Rector. *Rector - Instant Upgrades and Instant Refactoring of any PHP 5.3+ code*. 2023. Available also from: https://github.com/rectorphp/rector. Accessed on May 5th, 2023.

161. RECTOR. *New in Rector 0.15: Complete, Safe and Known Type Declarations*. 2023. Available also from: https://getrector.com/blog/new-in-rector-015-complete-safe-and-known-type-declarations. Accessed on May 5th, 2023.

162. SORBET. *Runtime*. 2021. Available also from: https://sorbet.org/docs/runtime. Accessed on May 7th, 2023.

163. MICROSOFT. *TypeScript Official WebPage*. 2023. Available also from: https://www.typescriptlang.org. Accessed on May 11th, 2023.

164. TOBIN-HOCHSTADT, Sam; ST-AMOUR, Vincent; DOBSON, Eric; TAKIKAWA, Asumu. *The typed racket guide*. 2014.

165. MAIDL, André Murbach; MASCARENHAS, Fabio; IERUSALIMSCHY, Roberto. Typed Lua: An optional type system for Lua. In: *Proceedings of the Workshop on Dynamic Languages and Applications*. 2014, pp. 1–10.

166. TEAM, R Core. *What is R?* 2023. Available also from: https://www.r-project.org/about.html. Accessed on May 7th, 2023.

167. STUDIO, R. *R Markdown*. 2023. Available also from: https://rmarkdown.rstudio.com. Accessed on May 10th, 2023.

168. TEAM, R Core. *The R Foundation*. 2023. Available also from: https://www.r-project.org/foundation. Accessed on May 7th, 2023.

169. TEAM, R Core. *R Language Definition*. 2021. Available also from: https://stat.ethz.ch/R-manual/R-devel/doc/manual/R-lang.html. Accessed on March 25th, 2023.

170. TEAM, R Core. *R language docs - matrix*. 2023. Available also from: https://rdrr.io/r/base/matrix.html. Accessed on May 8th, 2023.

171. WICKHAM, Hadley. *Advanced R*. CRC press, 2017. Available also from: https://adv-r.hadley.nz/s3.html. Accessed on March 23th, 2023.

172. TEAM, R Core. *R language docs - typeof*. 2023. Available also from: https://rdrr.io/r/base/typeof.html. Accessed on May 8th, 2023.

173. TEAM, R Core. *R language docs - numeric*. 2023. Available also from: https://rdrr.io/r/base/numeric.html. Accessed on May 8th, 2023.

174. TEAM, R Core. *R language docs - double*. 2023. Available also from: https://rdrr.io/r/base/double.html. Accessed on May 8th, 2023.

175. TEAM, R Core. *R language docs - integer*. 2023. Available also from: https://rdrr.io/r/base/integer.html. Accessed on May 8th, 2023.

176. TEAM, R Core. *R language docs - complex*. 2023. Available also from: https://rdrr.io/r/base/complex.html. Accessed on May 8th, 2023.

177. TEAM, R Core. *R language docs - logical*. 2023. Available also from: https://rdrr.io/r/base/logical.html. Accessed on May 8th, 2023.

178. TEAM, R Core. *R language docs - character*. 2023. Available also from: https://rdrr.io/r/base/character.html. Accessed on May 8th, 2023.

179. TEAM, R Core. *R language docs - list*. 2023. Available also from: https://rdrr.io/r/base/list.html. Accessed on May 8th, 2023.

180. *The Comprehensive R Archive Network*. [N.d.]. Available also from: https://cran.r-project.org. Accessed on March 13th, 2023.

181. WRENN, John; PAL, Anjali; VANHATTUM, Alexa; KRISHNAMURTHI, Shriram. *Dependently Typing R Vectors, Arrays, and Matrices*. 2023. Available from arXiv: 2304.04265 [cs.PL].

182. WEISSMANN, Ben. Typing the untyped: Soundness in gradual type systems. In: *The Strange Loop Conference*. 2019. Available also from: https://www.thestrangeloop.com/2019/typing-the-untyped-soundness-in-gradual-type-systems.html.