DSV Report Series No. 25-003

Stockholm University

On Typability in Programming Languages

Beatrice Åkerblom

Doctoral Thesis in Computer and Systems Sciences at Stockholm University, Sweden 2025

On Typability in Programming Languages

Beatrice Åkerblom

Academic dissertation for the Degree of Doctor of Philosophy in Computer and Systems Sciences at Stockholm University to be publicly defended on Thursday 20 March 2025 at 13.00 in Earth, Borgarfjordsgatan 12, Kista.

Abstract

This dissertation examines the relation between expressivity in programming languages and typability of programs.

For all data used in computer programs, it makes sense to perform a certain set of operations on it. The set of meaningful operations vary between different kinds of data, and this defines the type of the data. This means that all data used in computer programs has a type, but programs implemented using different languages treat information about types in different ways. The type information is accessible at different times throughout program implementation and execution.

In statically typed languages the type of all data is checked during compile time. This can provide programmers with valuable early detection of errors but can also lead to the rejection of programs that would actually run without any errors. In dynamically typed languages the approach is instead to defer checking if operations performed on some data is valid to run time. These checks are typically made at the the point in time when the operation is about to be performed. This allows an informed programmer to run programs that a static type system could not guarantee would be free from errors.

There is a trade-off between expressivity and e.g., early error detection and the possibility to use different kinds of optimisation techniques. This dissertation approaches the trade-off problem from several angles. We have examined existing programs to find out how code is written when there are no constraints from static types, both for the use of polymorphism in the dynamically typed language Python where there are no static typing at all, and access patterns used for arrays in the statically typed language Java. The latter has been used to evaluate the expressivity of array capabilities, a novel technique for statically preventing data races in parallel algorithms manipulating arrays.

Keywords: *Type systems, programming languages.*

Stockholm 2025
http://urn.kb.se/resolve?urn=urn:nbn:se:su:diva-238649

ISBN 978-91-8107-100-9
ISBN 978-91-8107-101-6
ISSN 1101-8526

Department of Computer and Systems Sciences

Stockholm University, 164 07 Kista

ON TYPABILITY IN PROGRAMMING LANGUAGES

Beatrice Åkerblom

Stockholm
University

On Typability in Programming Languages

Beatrice Åkerblom

ISBN print 978-91-8107-100-9
ISBN PDF 978-91-8107-101-6
ISSN 1101-8526

Printed in Sweden by Universitetsservice US-AB, Stockholm 2025

Abstract

This dissertation examines the relation between expressivity in programming languages and typability of programs.

For all data used in computer programs, it makes sense to perform a certain set of operations on it. The set of meaningful operations vary between different kinds of data, and this defines the type of the data. This means that all data used in computer programs has a type, but programs implemented using different languages treat information about types in different ways. The type information is accessible at different times throughout program implementation and execution.

In statically typed languages the type of all data is checked during compile time. This can provide programmers with valuable early detection of errors but can also lead to the rejection of programs that would actually run without any errors. In dynamically typed languages the approach is instead to defer checking if operations performed on some data is valid to run time. These checks are typically made at the the point in time when the operation is about to be performed. This allows an informed programmer to run programs that a static type system could not guarantee would be free from errors.

There is a trade-off between expressivity and *e.g.*, early error detection and the possibility to use different kinds of optimisation techniques. This dissertation approaches the trade-off problem from several angles. We have examined existing programs to find out how code is written when there are no constraints from static types, both for the use of polymorphism in the dynamically typed language Python where there are no static typing at all, and access patterns used for arrays in the statically typed language Java. The latter has been used to evaluate the expressivity of array capabilities, a novel technique for statically preventing data races in parallel algorithms manipulating arrays.

List of Papers

The following papers, referred to in the text by their Roman numerals, are included in this thesis.

PAPER I: **Tracing Dynamic Features in Python Programs**
Åkerblom Beatrice, Stendahl Jonathan, Tumlin Mattias, Wrigstad Tobias, *Proceedings of the 11th Working Conference on Mining Software Repositories*, **MSR 2014**, Pages 292-295 (2014).
DOI: 10.1145/2597073.2597103

PAPER II: **Measuring Polymorphism in Python Programs**
Åkerblom Beatrice, Wrigstad Tobias, *ACM SIGPLAN Notices*, **Volume 51, Issue 2**, Pages 114-128 (2016).
DOI: https://doi.org/10.1145/2816707.2816717

PAPER III: **Reference Capabilities for Safe Parallel Array Programming**
Åkerblom Beatrice, Castegren Elias, Wrigstad Tobias, *The Art, Science, and Engineering of Programming*, **Volume 4, Issue 1**, Article 1 (2019).
DOI: 10.22152/programming-journal.org/2020/4/1

PAPER IV: **Progress Report: Exploring API Design for Capabilities for Programming with Arrays**
Åkerblom Beatrice, Castegren Elias, Wrigstad Tobias, *Proceedings of the 14th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, **ICOOOLPS '19**, Pages 1-8 (2019).
DOI: 10.1145/3340670.3342427

PAPER V: **Arrays in Practice: An Empirical Study of Array Access Patterns on the JVM**
Åkerblom Beatrice, Castegren Elias, *The Art, Science, and Engineering of Programming*, **Volume 8, Issue 3, Article 14**, page (2024).
DOI: 10.22152/programming-journal.org/2024/8/14

Contents

List of Figures

List of Tables

1. Introduction

When writing programs, different tools are used to secure that the written code matches the the intended functionality: testing tools, debuggers, etc. It is also common that the programming language provides a type system that is used to check that all operations performed by the program are meaningful. The type system defines the types for a language, and the values used in a program are categorised to belong to one of its valid types. The types, in turn, define how operations available in the language can be applied on values of these types, and makes it possible to perform analyses on the program before it is run.

The goal of this work has been to study *typability* from different angles. Typability is the ability to find a precise type for a given expression, a type that is specific enough to help the programmer or compiler in analysing the code, but still flexible enough to be useful in practice. To achieve this goal I have examined how types are used in real-world programs written in dynamically typed programming languages, and also how types can be developed further to provide the programmer with extended support for alias control in concurrent and parallel programs.

In the end the aim of this work is to contribute to the creation of programming languages that use types to support programmers when the support is needed, while still allowing programmers the freedom to use their skills to write code that may not be possible to type. That is writing working code for which a type system cannot prove the absence of errors.

1.1 Research Question

The overall research question examined in this research project is "To what extent can we use static types to capture dynamic behaviour?", and the work has been conducted on two separate themes to understand and improve typability:

Theme I: Understanding how types are used in programs written in dynamically typed languages On the first theme, I have conducted two empirical studies to understand typability by examining to what extent programs written in dynamically typed languages actually utilise the dynamicity that the language allows. Dynamic languages typically allow the use of highly dy-

namic features such as redefinition of classes (and even individual objects) during run time and unconstrained polymorphism, that is allowing the same code to operate on different types without any restrictions (see further discussion in Section 3.1.2). In this context I have explored the possibilities to retrofit programs written in dynamically typed languages with static types. The studies were made on Python programs to answer some more specific research questions related to the theme:

- Do programmers actually use dynamic features available in dynamic programming languages that make changes to types possible at run time?

  I have studied programs to find out how often, when, and where in the code language constructs that make changes to existing types (or perform other dynamic operations) are used, see Paper I.

- Do programmers write programs using polymorphic variables?

  I have studied programs to investigates how common it is that the type of variables receiving method calls are polymorphic when the same code section is executed at different times throughout program runs, see Paper II.

Theme II: Improving support for programming with arrays On the second theme, I have used both theoretical and empirical methods to develop and evaluate more precise types for arrays that support the programmer by preventing race-conditions in concurrent and parallel programming. In the theoretical work, I have presented *array capabilities*, an array type that supports alias control and guarantee thread-safe concurrent operations on arrays. The empirical work was made to allow further development of the theoretical work by examining how arrays are actually used in real-world programs and by implementing a prototype of array capabilities to allow practical evaluation. The specific research questions based on the second theme were:

- How can concurrent programming patterns involving arrays be checked statically?

  I have presented array capabilities, an array type supporting operations for splitting arrays into sub-arrays. I have proven that distributing the sub-arrays over several threads will never lead to race-conditions, see Paper III.

- Are array capabilities useful in practice?

  I have implemented a prototype of the array capabilities and evaluated it by using it to implement common array algorithms, see Paper IV.

- Are arrays used in regular patterns that allow splitting according to the operations defined for array capabilities?

  I have studied programs to investigate the regularity of array accesses, see Paper V.

1.2 Contributions

This thesis consists of a collection of papers that cover different aspects of typability. In this section I explain the research gaps that the papers aims to address and the contributions made by each of them.

On the first theme, **Understanding how types are used in programs written in dynamically typed languages**, the studies made for Paper I and Paper II were motivated by the large interest shown at the time for developing type systems for dynamic programs [1–9]. Some efforts had been made to investigate the difficulties faced if retrofitting a static type system on existing code written in dynamic languages, *e.g.*, for Javascript [10; 11], for Smalltalk [12], and for Python [13].

The study of the use of dynamic features in Paper I is an extension and improvement of the study made by Holkner and Harland [13], which suffered from performance penalties and unmanageably large logs. The results confirm and extend their results.

My contributions are that I show that the use of language constructs that potentially make changes (*e.g.*, by adding or removing methods) to existing types is commonly found in Python programs. Their presence is sometimes sparse, but they are found in both library code and program-specific code, at program start-up and throughout the entire run time. This leads to the conclusion that it is common that Python programs make use of dynamic features to exhibit behaviours that are inherently hard to type.

The study of polymorphism made for Paper II is the first empirical study specifically of polymorphism in programs written in dynamically typed languages.

By studying the actual types present at run time I could show that most call-sites are monomorphic and thus that large parts of programs can be typed using a simple type system. The polymorphic call-sites of the programs, however, can to a great extent not be typed using nominal or structural systems.

On the second theme, **Improving the support for programming with arrays**, Paper III, provides a solution for how to combine reference capabilities with arrays, which was unaccounted for [14–16].

Paper III presents *array capabilities*, an extension to reference capabilities that allows programmers to split arrays into sub-arrays with a statically guar-

anteed absence of data races, since no array elements will be part of more than one sub-array unless the array can only be read from and never written to.

Paper IV presents the evaluation of a prototype implementation of the ideas presented in Paper III which allowed me to show that by using *array capabilities*, programmers may express common array algorithms without explicit use of indexes. Since part arrays can be treated as arrays, the algorithms may be expressed at a higher level of abstraction.

Static analyses of array access patterns are used for program optimisation and program transformations at compile time to ensure that array accesses are independent and will not affect each other [17–21]. Paper V presents the first dynamic study of array access patterns, which produces the exact indexes accessed when the programs were run. Exact indexes are expensive, and in general undecidable, to calculate statically.

The study presented in Paper V was made to get further input in our continued work on array capabilities from Papers VI, III and IV, by providing information about how arrays are used in real-world programs.

This section continues with a summary of the contributions of each paper.

Paper I: Tracing Dynamic Features in Python Programs

In paper I [22], we present the results from studying the use of some highly dynamic built-in language features from Python. The features include functionality to dynamically introspect, add or remove attributes, and functionality for dynamic code generation and dynamic library loading. Programs using most of these dynamic features (introspection excluded) can not be statically typed in general, since their properties will not be known until run time. Using flow/dependency information would allow capturing additional interesting cases statically, but type analysis is traditionally flow-insensitive.

The study is based on run time traces from 19 open source programs with a graphical user interface implemented entirely in Python. The traces were collected by running the programs in an instrumented version of the CPython interpreter for Python 2.6.6.

We make the following contributions:

- We increase the understanding of actual dynamic program behaviour in real-world Python code in terms of what dynamic features are used, whether they are used in program-specific code or library code, and during start-up or normal execution.

- We develop tracing additions to an existing Python interpreter which circumvents the performance penalties and unmanageably large logs of

previous work [13], which hampered the validity of previously available results.

The results show that Python programs do leverage built-in dynamic features to exhibit behaviours which are inherently hard to type. All programs in the study use many (between 5 and 9 out of 15) of the included dynamic features. All programs use some dynamic feature from each of the categories (introspection, object change, dynamic code generation and dynamic library loading). The extent of this usage of each feature in the different programs was extremely varied, ranging from 2 uses to over 5 million uses.

The findings can be summarised as follows:

- The use of dynamic features was found both in library code and program-specific code. This rules out a solution for introducing static types where libraries (assumed to be more stable) are statically typed and used from dynamically typed program-specific code.
- Dynamic behaviour is found both at program start-up and throughout the entire run time, which mean that restricting use of highly dynamic program constructs once programs "stabilise" is not possible.

From a typability perspective, the results from this study show that we need to allow programs written in dynamically typed languages the possibility to remain dynamic both in library code and in program-specific code and throughout the run time.

Paper II:
Measuring Polymorphism in Python Programs

In paper II [23], we present the results of a trace-based study of 36 open-source Python programs, examining the run time types of the variables receiving method calls and additional run time information about these method calls.

The programs included were all stand-alone programs (both GUI programs and command-line programs) entirely implemented in Python. The traces were collected by running the programs in an instrumented version of the CPython interpreter for Python 2.6.6.

The study is concerned specifically with typability in that:

- We examine how types at polymorphic call-sites relate to each other in terms of inheritance and overridden methods
- We examine to what extent is it possible to find a common super type for all the observed receiver types that makes it possible to fit the polymorphism into a:

- *nominal static type* (see Section 3.1.2)–that is do all receivers share a common super type that defines the called method?
- a nominal static type if extended with *parametric polymorphism* (see Section 3.1.2)–if the receiver types were grouped by the object identity of the sender, can we find a common supertype for each group that contains the methods called at the call-site?
- structural type (see Section 3.1.2)–that is to what extent do receiver types in code blocks contain all the methods that are called at the call-sites?

The results show that variables that act as receivers of method calls are predominantly monomorphic, although most programs contain a few highly polymorphic call sites (method calls made on variables containing values of many different types at different times, in different contexts). An approximation is made of the extent to which the studied programs could be typed using nominal types, nominal types with parametric polymorphism, and structural types.

Our findings can be summarised as follows:

- As a consequence of the extensive monomorphism, most programs can be typed to a large extent using simple type systems.
- However, most polymorphic and highly polymorphic parts of programs could not be typed using nominal or structural type systems, for example due to use of value-based overloading. For example if the value of an argument is being used to control the execution path of a method.
- Structural typing is only slightly better than nominal typing at handling the non-monomorphic program parts.

Paper III: Reference Capabilities for Safe Parallel Array Programming

In paper III [24], we present array capabilities, an extension of reference capabilities [25] that support race-condition-free concurrent operations on arrays of both primitive and non-primitive values.

The array capability is an abstraction that stores a physical array, information about the length of the current array capability (shorter or equal to the underlying array) and a translation function for the indexes to allow reordering of the stored elements. A goal of the array capabilities is to allow array algorithms to operate at a higher level of abstraction, where parts of arrays and arrays can be treated equally without keeping track of start and stop indexes.

The array capability enables splitting arrays into sub-arrays logically by storing the same physical array together with different lengths and translation functions. A language of operators for *e.g.*, splitting and merging array capabilities and for physically aligning the physical order in the underlying array with the logical order defined by the translation function. Explicit merging is available in the calculus, but it is often not needed since implicit merges are used on capabilities when leaving a scope and returning to the point where a split was made.

We make the following contributions:

- We formalise the dynamic semantics of the key operations on array capabilities in a core calculus.
- We prove type soundness and array disjointness (that two capabilities that allow mutation never give access to overlapping sets of elements). Data-race freedom is stated as a corollary to these theorems.
- We introduce array capabilities, an extension of reference capabilities, that provides built-in support for subdivision of arrays into disjoint parts to enable data-race free parallel programming with support for arrays.
- We examine how array capabilities can support abstract manipulation of arrays through logical splitting into subarrays and merging of subarrays

Array capabilities extend the safety of reference capabilities to arrays in a way that build cleanly on existing reference capability concepts and constructs. This allows programmers to express parallel array algorithms at a higher level of abstraction, where parts of arrays can be treated as arrays without keeping track of the parts' start and stop index, and with a statically guaranteed absence of data races.

From a typability perspective, array capabilities extend the safety of reference capabilities to arrays. This aims to allow programmers to express parallel array algorithms at a higher level of abstraction, while absence of data races is guaranteed statically.

Paper IV:
Progress Report: Exploring API Design for Capabilities for Programming with Arrays

Paper IV [26] expands and validates paper III by presenting and discussing a prototype implementation of the language introduced there.

The prototype implementation was made using Python, a language without a static type system. This allows the focus of this paper to be on the exploration of the API design and questions like what the frequent operations on

array capabilities in typical parallel array algorithms are; which fundamental operations can be used to construct most other operations and how can one expose these operations to programmers.

The findings can be summarised as follows:

- Two new splitting operations were identified and added to the API for convenience.
- The splitting and merging operations from Paper III (consecutively and in strides) were used in many of the example algorithms which validates their inclusion.
- The importance of the merge operation was found to be less important than was initially thought.
- The dependence on direct index based accesses decreased in general by using the prototype implementation.

Paper V:
Arrays in Practice: An Empirical Study of Array Access Patterns on the JVM

In paper V [27], we report on the study of arrays made to gain a better understanding of how arrays are used in a collection of real-world programs. Using a dynamic, trace-based approach, the programs were first instrumented to log information about all array accesses, and then the programs were run.

We make the following contributions:

- We present a method for capturing general characteristics of arrays together with all accesses made to them running on the JVM.
- We present a strategy for detecting patterns and sub-patterns in traces of array accesses.
- We show that, on average, 69.8 % of the array access patterns for each benchmark consist entirely of combinations of uncomplicated sequential traversals and repeated accesses to the same index.

The collected data was analysed for various characteristics of array usage. The following questions were investigated:

- What are the characteristics of arrays created and used in real-world programs? This includes looking at metrics such as array sizes, what data types are stored in arrays and how large parts of arrays are actually accessed.

- From where are arrays accessed? This incorporates collecting data about where accesses are made from, tracing both from what classes the accesses are made and the identity of all threads that access an array.
- Are arrays accessed in regular patterns? Answering this question requires studying the distribution of all accesses to individual arrays to discover emerging patterns, comparing the access distributions between arrays to find arrays that have been accessed in the same way, but also to find arrays that have been accessed in a unique way, and measuring the proportion of the arrays where arrays are made in regular traversals.

The methodology used in this study can be applied to any program running on the Java Virtual Machine. The results from the study and the methodology used, can inform future run time implementations and compiler optimizations.

The findings can be summarised as follows:

- Array sizes are mostly small. 76.8 % of all arrays contain 10 or fewer elements.
- Most arrays are accessed by only one class (75.3 %), and most arrays are accessed from one single thread (97.2 %).
- A majority (69.8 %) of all array access patterns consist of combinations of uncomplicated sequential traversals and repeated accesses to the same index.
- More than half (53.8 %) of all individual array accesses are made as parts of identifiable uncomplicated sequential traversals or constant sequences where the same index is accessed repeatedly.

The Author's Contributions

PAPER I: Article written together with author four. I did all the work with instrumenting the Python interpreter. The data collection was done by author two and three.

PAPER II: I was the main author of this article which was written together with second author. I did all the work with instrumenting the Python interpreter, the data collection and the data analysis.

PAPER III: I was the main author of this article which was written together with author two and three. Semantics and proofs created in collaboration with author two.

PAPER IV: I was the main author of this article. Sole implementor.

PAPER V: I was the main author of this article written together with second author. I did all implementation work, the data collection and the data analysis.

The articles on Python were published in 2014 and 2015, respectively, and a lot has happened in Python and other dynamic languages since then. A discussion about this can be found in section 4.1.5.

Related Article

PAPER VI: **Parallel Programming with Arrays in Kappa**
Åkerblom Beatrice, Castegren Elias, Wrigstad Tobias, *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, **ARRAY 2018**, Pages 24-33 (2018).
DOI: 10.1145/3219753.3219757

This paper motivates and introduces array capabilities, a new kind of reference capability to use with arrays. This work is continued in Paper V.

1.3 Summary of Conclusions

To answer the he overall research question "To what extent can we use static types to capture dynamic behaviour?", I have conducted work on two separate themes to understand and improve typability.

The conclusions that I have drawn on the first theme, "Understanding how types are used in programs written in dynamically typed languages", are in summary that programs written in dynamically typed languages can not entirely be retrofitted with a static type system. However, programmers write code that is to a great extend typeable even when they are not forced to by a static type system. This strengthens the case for gradual typing, where parts of a program could be extended with static types while other, more dynamic, parts of the same program are left without static types.

The conclusions drawn on the second theme, "Improving support for programming with arrays", are in summary that static types can be extended to support programmers in writing parallel programs operating on arrays.

As a whole, the conclusion is that we can use static types to capture dynamic behaviour to a great extent, but not entirely. Types can be used to provide programmers with more support when writing complex code, *e.g.*, in ar-

ray algorithms, but the language should also allow the programmer to write programs that contain behaviour that cannot be captured by static types.

1.4 Outline

The rest of this overview and summary of the work done for this dissertation is structured as follows:

- Chapter 2 presents the research methods used in the papers included in this dissertation.
- Chapter 3 provides the necessary background on types in statically and dynamically typed languages. It also contains explanations and motivations for why the array is a data structure of special interest for typability.
- Chapter 4 presents the results achieved and conclusions made about typability through the papers included in this dissertation.
- Chapter 5 concludes and discusses possible directions for future work.

2. Research Methods

In producing the work presented in this dissertation I have applied different research methods common in programming language research both to develop new language features, and to analyse and evaluate how existing programming language features are used in practice. The methods used include both empirical and theoretical methods.

2.1 Empirical Methods

Three of the papers included, Paper I, Paper II, and Paper V, first of all aim at establishing an understanding of how real-world programs function in some specific way using a quantitative method. This involved the use of empirical methods to collect data.

2.1.1 Dynamic Trace-Based Studies

Common for Paper I and Paper II is that the information needed is not available in the code but only when the programs are running, which rules out static analysis. For Paper V, static analysis (as used *e.g.*, in [17–21]) is a possible option, but the result would have been less precise and not complete due to the complexity of calculating the array indexes. To make the data collection possible for Papers I and II, and to get a more precise result for Paper V, a dynamic trace-based approach [28] was used, where the data is output from the running program itself. The result is a structured observation of certain aspects of the running programs.

To select programs for inclusion in the studies we used a survey strategy. The programs studied in Paper I and Paper II were collected from Source Forge [29], which was a major source for open source programs at the time when the studies were made. In Paper V, the study included all programs from the Renaissance Benchmark suite [30].

To enable data collection from the running programs, either the run time environment or the programs themselves must be instrumented to output the relevant information. For Paper I and II, the instrumentation was made in the Python interpreter and for Paper V, the instrumentation was made in the class

files of the Java programs. Since the data collected will be dependent of how the program was run and the specific execution paths taken in the program, the logs from different runs of a program may differ. In the data collection for Paper I and Paper II, care was taken to use all functionality of the programs *e.g.*, by using all available menu alternatives in programs with a graphical user interface. For Paper V, the program execution was performed by running the benchmarks.

This approach to data collection, dynamic and trace-based, comes with advantages and disadvantages compared to static analyses based on source code. First of all, it easily captures events (*e.g.*, the use of a language feature of interest) in code which is loaded or even generated at run time. Generated code is not present in the source code and therefore it may not be captured by static analysis. Furthermore the running program will for example have access to values and more specific information about *e.g.*, types than available in the source code. This kind of dynamic tracing of programs is however sensitive to different paths taken in a program due to input. The main disadvantage of the dynamic approach is the difficulty of choosing representative input or interaction strategies for running programs which will give acceptable code coverage.

The quantitative data collected in the traces was evaluated and analysed through numerical and statistical measurements.

2.1.2 Prototyping

Prototyping is a method commonly used in the field of human-computer interaction and design. A prototype implementation can be allowed to focus on certain qualities while allowing omission of details that must be included in a final product [31].

For Paper IV, a prototype version of the array capabilitities proposed in Paper III was developed and evaluated. The prototype, implemented in Python, allowed us to omit the actual implementation of the type system and focus on the API design.

The evaluation of the prototype was performed by using it to implement typical array algorithms. The resulting code was compared to pure Python implementations and the comparisons made were both quantitative (*e.g.*, by comparing the number of direct index manipulations needed) and qualitative (*e.g.*, noting the decreased importance of certain API components in some situations). Similar evaluations, but at a much larger scale has been conducted to evaluate extensions to C# [15]. In that case the language has been used for writing millions of lines of code in an entire in-house project at Microsoft. Our evaluation aimed at the same goal, but in smaller circumstances.

2.2 Theoretical Methods

Theoretical methods were used in Paper III (see Section 1.2) to define and prove properties about new programming language features.

The language features were presented formally by its grammar, static semantics and dynamic semantics. The semantics were presented using operational semantics. Another possibility would have been to use denotational semantics. The result is modular, abstract, and mathematically elegant but the need for fixed-point analysis for loops and recursive functions makes it cumbersome to use. Operational semantics provide a more straightforward model with concrete step-by-step execution of programs.

Formal proofs were used to demonstrate language properties such as type safety, and important properties of the new language features introduced.

2.3 Ethical Considerations

The data for the empirical studies in Paper I, Paper II and Paper V, was in all cases collected from programs from the open source domain. No ethical approval was needed.

2.4 Societal Consequences

The work presented in this dissertation aims to gain a better understanding for programming languages so that future compilers and run time systems can be made more efficient and provide better support for programmers when writing code.

The results from Paper I and Paper II are useful when developing support for static types in dynamically typed languages. The results from Paper III and IV are useful in creating parallel languages supporting the programmer when using arrays. The results from Paper V are useful in compiler construction and as input for further work on array abstractions like the one presented in paper III.

3. Background and Related Work

The work presented in this thesis belongs to two separate main areas. All papers present work examining types and typability and Papers I and II do this in the setting of dynamically typed languages and Papers III-V focus on typability when it comes to arrays. This chapter gives the relevant background for both of these areas.

3.1 Types and Type Systems

Computer programs are used to store many different kinds of data and to perform computations on many different kinds of data. The computer's storage, however, does not distinguish between different kinds of data. Early programming languages did, as very low-level programming languages (*e.g.*, assembly languages) still do today, require the programmer to keep track of what kind of data has been stored and what operations could be used on that particular data in a meaningful way. In more recent and higher level languages the programmer gets support in monitoring the use of data by using types.

This section gives an overview of types and type systems and how types can be used in different ways in different programming languages. Section 3.1.2 discusses types in statically typed languages which aim for type safety and early error detection. The requirements for book-keeping types in the code may however be perceived as getting in the way of expressing clear and concise solutions to programming problems. Section 3.1.4 discusses dynamically typed languages which, on the other hand, are appreciated by many for the expressiveness, but fall short in *e.g.*, helping programmers finding errors. Section 3.1.5 discusses combining static and dynamic typing using different kinds of gradual typing aiming to benefit from the advantages of both static and dynamic typing at the same time. Section 3.1.6 finishes off by discussing typability.

3.1.1 Types

A type is a description of some characteristics of a value. There are a number of common perspectives for understanding types: denotational, structural,

and abstraction-based [32]. In the context of the work presented in this dissertation, the abstraction-based perspective is the most useful one, where a type describes the inventory of methods available to call in an object as an interface of communication. The type sets the rules for how an object can be used in a meaningful way, that is what operations we are allowed to perform on it. This means that the type is an abstraction of the language's dynamic semantics.

The first generations of programming languages that used type information to analyse code only provided built-in types, *e.g.*, early versions of Fortran [33] that originally only supported integer and floating point numbers. A slightly extended set of built-in types (typically including characters and Booleans) are provided also by most modern languages. Composite types like arrays and lists [34] were also included in early languages. The possibility to build more complex types composed by using existing types in a `STRUCT` was introduced by Algol [35].

Pointer types were introduced in PL/I [36][1] to increase flexibility, *e.g.*, for operations on linked lists. Instead of storing a saved value directly in a variable, a pointer variable stores the memory location of where the value is located. The flexibility gained by using pointer types, *e.g.*, the possibility to share a value without copying but only pass the information about where it can be found (the pointer), comes with increased complexity and introduces the aliasing problem, discussed further in Section 3.1.3.

Later, through the introduction of abstract data types [37; 38], it became possible to extend user-defined types with tailored operations. The abstract data types defined by the programmer both enables the compiler to perform more advanced analyses to ensure program correctness and provided a way to let programmers make use of data collections without having to understand the internal details of their implementation. A good example of such use is *iterators* which allows access to all elements in a collection without exposing how the elements are stored internally [32]. The idea of abstract data types has been adopted in a wide range of languages and is a cornerstone in class-based object orientation [39].

Statically typed languages require that values are associated with a type at compile time. Commonly, the programmer must explicitly express the intended type at certain points in the program *e.g.*, variable and function declaration) by annotating the program's variable and subroutine definitions with intended types (as in *e.g.*, Java, C# etc.). Another option is to let the compiler infer the types of all expressions, either entirely as in *e.g.*, ML [40] and Haskell [41], or partially as when using the `var` construct in *e.g.*, C# [42] or Java [43]. In both cases there will be check points where the compiler can

[1]The Address Language, created in 1955 by Ukrainian Kateryna Yushchenko also used pointers, but was not known outside the USSR.

ensure that the expected types match the ones provided.

Types, when annotations are required, increase program readability [44] although the need to annotate the code with types may make programs somewhat harder to write [32]. Fundamental motives for type systems are helping programmers in detecting errors and enabling compiler optimisations. Over the years, type systems have evolved to find and diagnose a multitude of errors and support programmers *e.g.*, by improved code readability and documentation, in creating and using abstractions, and by enabling tool support [45]. Along with the evolution of type systems, the support for programmers to devise more flexible types (*e.g.*, by use of polymorphism) has also grown. Further, types are nowadays used in a wide range of more advanced code analyses, *e.g.*, contract-based verification [46], alias analysis [47], ownership types [48], deadlock and data-race prevention [49].

3.1.2 Static Type Systems

> "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."[45]

Types are the building blocks of type systems that can be used to prove absence of unwanted behaviour. If type correctness can be proven statically we can rule out several categories errors before even running the program [44].

A type system defines the types for a language and how the operations available in the language (the language constructs) are related to the types available. It also sets the rules for determining if types are equivalent or compatible and in some cases how a type can be inferred.

An important goal for the use of type systems to analyse programs is to achieve type safety, that is avoiding type errors at run time. Enabling compiler optimisations and helping programmers in detecting errors are still fundamental motives, but type systems have since evolved and proven to be useful in a multitude of more advanced analyses as will be discussed below.

Subtyping

The Liskov Principle of Substitution [50] states that a type can always be substituted by a subtype, which means that program correctness will be preserved if a type `A` is replaced by a type `B` as long as `B` is a subtype of `A`. Subtyping can be approached in two general ways, either *nominal* (used in *e.g.*, Java, C#, and Pascal) or *structural* (used in *e.g.*, PL/I Algol, and Modula-3). Many languages use combinations of the two approaches.

When using a nominal approach, subtyping requires that the subtype explicitly declares the relation in its definition. An example of such a declaration expressed in Java can be found on line 10 in Listing 3.1, where the class `B` is declared to be an extension (subtype) of class `A`. Java will, given these circumstances, allow the object of `class B` created on line 2 to be assigned to the the variable `b` of type `A` on the same line. This means that the static type (the type in the code) of `b` is `A`, while the run-time type will be `B`. Allowing a subtype to substitute a super type means that the value stored in `b` may have "many forms". This is called *subtype polymorphism*. Subtyping is a one-way relation and type `A` can generally not be used to replace type `B`.

In an object-oriented language like Java the subtype will inherit all methods from its supertype, which means that all operations that can be performed on a supertype will be available also in the subtype. A subtype may add methods and change the implementation of the operation, as is done for the method `getMessage()` in the class `B` on lines 10-14 in Listing 3.1, but the methods available in the supertype will always be available in the subtype. Subtype polymorphism can also be achieved by using interfaces [51], but the result is similar.

```
1  public static void main(String[] args) {
2      A b = new B();
3      System.out.println(b.getMessage());
4  }
5  static class A {
6      public String getMessage() {
7          return "I'm an A object";
8      }
9  }
10 static class B extends A {
11     public String getMessage() {
12         return "I'm a B object";
13     }
14 }
```

Listing 3.1: Code example showing the use of nominal subtyping in Java.

Using a structural approach instead omits the requirement for explicit type relations in the class declaration. Type equivalence is based on the actual structure of the types, usually the set of instance variables and methods available but the exact requirements varies somewhat between languages. In the code example in Listing 3.2 we see two classes, `Point` and `Coordinate`, that are structurally identical. They contain the same number of instance variables, with the same types and even the same names. Their constructors and methods

```
1  public class Point {
2     private int x;
3     private int y;
4
5     public Point(int x, int y) {
6        this.x = x;
7        this.y = y;
8     }
9     public void move(int xDistance, yDistance) {
10       this.x += x;
11       this.y += y;
12    }
13 }
14 public class Coordinate {
15    private int x;
16    private int y;
17
18    public Point(int x, int y) {
19       this.x = x;
20       this.y = y;
21    }
22    public void move(int xDistance, yDistance) {
23       this.x += x;
24       this.y += y;
25    }
26 }
```

Listing 3.2: Code example showing two classes (types) that could be regarded as equivalent if using a structural approach, which Java does not support.

also take the same number of arguments of the same types and the return value of the `move()` method is `void` in both cases. This means that an object of type `Point` could safely be used instead of an object of type `Coordinate`, which would be allowed if type equivalence is based on structure.

Structural types are more flexible. It is for example possible to change the name of a type without having to find and change that name in all possible subtype declarations as would be necessary using nominal types. On the downside of structural types, however, is that types that have nothing in common may be considered to have a supertype-subtype relation. A common example to illustrate this is to use a type `Circle` and a type `Cowboy`, both with a method `draw()` that will have disparate functionality but would let `Cowboy` be a subtype of `Circle` in a structurally typed setting. The explicit definitions of a nominal approach has an advantage given these premises in that it reveals the programmer's intent.

```
1  class Parametric {
2     public static void main(String[] args) {
3        B b1 = new B<A>(new A());
4        B b2 = new B<C>(new C());
5     }
6     static class B<T> {
7        private T f;
8        public B(T f) {
9           this.f = f;
10       }
11    }
12    static class A { }
13    static class C { }
14 }
```

Listing 3.3: Code example showing a class `B` uses parametric polymorphism to enable the instance variable `f` to hold instances of different classes in different instantiations.

Another category of polymorphism is the *parametric polymorphism* where *e.g.*, a class can be written to expect type information as a parameter on object creation. The code in Listing 3.3 contains a class `B`, declared on line 6-11, which is "incomplete" but can be instantiated if a type `T` is given as a parameter on object creation. The type variable `T` will then be replaced by the type given on instantiation. On line 3, the instantiation is done with the type `A` and on line 4 the instantiation is done with the type `C`. In both cases a new object of the type used as a parameter is passed as an argument to the constructor which stores the object in the instance variable `f` (on line 9).

An image of the results from running the code in Listing 3.3 is shown in Figure 3.1. Although there is no subtype relation between the type `A` and the type `C`, the instance variable `f` is able to hold an object of type `A` in object `b1` and an object of type `C` in the object `b2`.

3.1.3 Static Types for Alias Management

In programming languages that uses pointers or references to access values stored in memory, the pointer or reference is a value in its own right. The reference can be copied and shared between different parts of the program, and in turn be stored in more than one location. When more than one reference point to the same value, they are *aliases* [52]. This means that any changes made through one variable may impact the value read through another. Aliasing increases flexibility and saves memory and computer power since data need not be copied, but makes programs more difficult to overview and understand

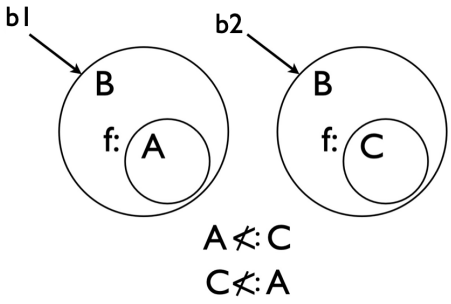


Figure 3.1: Parametric polymorphism. The two instances, `b1` and `b2`, of class `B` hold objects of type `A` and `C`, respectively, in their instance variables `f` even though no subtyping relation exists between the types `A` and `C`.

and has long been recognised as a cause of errors that are hard to detect [53]. Possible aliasing also rules out some optimisation techniques performed by compilers, which may result in slower programs [32].

Aliasing is possible in any language supporting references, but its use in procedural languages like Fortran, Algol, C etc., is often bound to a specific scope (*e.g.*, a method call) and thus the aliasing also ceases to exist when program execution exits that scope. This kind of *dynamic aliases* [54] are short-lived and less problematic than *static aliases* [54], that is aliases saved in a struct or in an object's instance variables when using object-oriented languages. In object oriented languages, objects' preservation of their internal state make static aliases long-lived and more problematic. Many solutions have been presented to deal with the aliasing problem.

In *The Geneva Convention On The Treatment of Object Aliasing* [53], the authors defines the aliasing problem in the context of object-oriented languages. Solutions to the problem are divided into four general categories; alias detection, alias announcement, alias prevention and alias control [53]. The work presented in this dissertation belongs to the alias prevention category and the main focus of this background has been set there.

Alias detection is described as "Static or dynamic (run time) diagnosis of potential or actual aliasing" [53]. Within this field, a lot of work has been done and is still being done in static pointer analysis [55; 56]. Static pointer analysis requires no annotations and gives no performance overhead, since the analysis is made at compile time, but due to to computational complexity the result will often be incomplete and an approximation. The resulting information can be used for optimisation or for error detection.

If the analysis is performed at run time, the result of the analysis may have higher precision, but it will produce a run time overhead that impacts the

program performance. Another obstacle for dynamic alias detection is that the result will depend on code coverage; aliases may be undetected if they occur in paths that were not executed. A tool that may be used for dynamic alias detection is for example Spencer [57].

Alias announcement is described as "Annotations that help modularize detection by declaring aliasing properties of methods" [53]. Due to the non-local nature of aliasing, aliases may be hard to detect by reading code. Alias announcement introduces programmer annotations that make it visible if *e.g.*, an argument reference to a method ends up captured in an instance variable or in other ways aliased, or if it to the contrary will never end up aliased.

Aliasing contracts [58] lets the programmer use annotations to specify preconditions [46] for accessing objects, which are checked at run time. The annotations prescribe how the actual object can be accessed, not how references to the objects are created, copied etc. The run time checking makes the system flexible. The contracts may depend on information about the actual aliasing situation in the program which is unavailable to a static analysis.

Alias control aims to provide "Methods that isolate the effects of aliasing" [53]. An effect system extends the type system with mechanisms that manages and checks the side effects resulting from running the code. An adaptation to the object oriented paradigm is to declare effects in terms of abstract *regions* to avoid exposing details about objects' internal representation [59]. Combined with some techniques from alias prevention *e.g.*, *alias burying* [60] or *ownership types* [48], described in the next section, an effect system is a flexible tool for managing aliasing [61; 62] even in parallel programming [63].

Alias prevention consists of "Constructs that disallow aliasing in a statically checkable fashion" [53]. These techniques require programmer annotations, *e.g.*, `noalias` which are used to statically check that aliasing will never occur. This approach will prevent a reference annotated as `noalias` from ever being copied unless the original is destroyed.

One example of this at a fine-grained level is to replace reference copying with *swapping* to avoid the creation of aliases while also avoiding the copying of potentially large data structures [64]. Swapping prevents aliases to ever occur but also requires programmers to change common programming patterns. When using swapping, for example to pass a value `x` as an argument to a subroutine `f(x)`, the subroutine will have to return the value to preserve any changes made to it, or the programmer has to explicitly create an alias to `x` (in [64] by calling a function `replica()`) and pass that alias as the argument.

Another example of a fine-grained solution is combining references that may only be used for reading with unique objects (that are always unshared) and *destructive reads*, which means that the original reference will always be set to `null` whenever its value is read [65]. The use of destructive reads

programs will however pollute programs with problematic `null` references [1], which can be avoided by instead using *alias burying* [60]. With burying, static analysis is used to ensure that a buried reference is never used again in the program, which allows copying a unique reference. In addition to burying, *alias borrowing* for dynamic aliases, *e.g.*, for values passed as arguments with a method call, can be used to remedy the tedious copying of references required by destructive reads [60]. Borrowing a reference is allowed as long as the borrowed reference is never stored on the heap (e.g., in some instance variable) and if the original reference cannot be used while the borrowed reference has not been returned.

The solutions presented above do provide alias prevention, but they are all impractical in real life programming both since the limitations for aliasing are too restrictive and because the introduction of `null` values after destructive reads is a source of errors. Even though it introduces problems, aliasing is an important feature in object-oriented languages without which it would be impossible to *e.g.*, implement common data structures such as doubly linked list.

By using *islands* [54] to enable the creation of boundaries around groups of objects that are related, aliases may be prevented from crossing these boundaries. This provides a solution that may be tailored to be more coarse-grained. The objects within an island may be accessed from the outside only through one single access point (a bridge object), while allowing the objects inside the island to hold aliases to each other in any way. The bridge object controls moving objects onto or off the island by a combination of references that can only be used for reading operations and `unique` references, destructive reads (i.e. swapping with null), and borrowing.

Almeida [67] points out that even though enforcing encapsulation is sometimes desirable, it is also sometimes preferable or even necessary to break it. Balloon types (that are only concerned with static aliasing [54]) are presented to handle this by supporting a differentiation between internal representation that may never be referenced by external objects, and external objects that may be referenced from anywhere.

Ownership types [48] are static types annotated with contexts, which indicates ownership. By introducing a hierarchy of nested contexts among the objects, where all objects belong to a context and in turn own a context (containing all objects that are regarded as belonging to that object's representation). If object `b` is owned by object `a` it is only possible to access `b` through object `a`. The number of references to `b` is not limited, as long as they come

[1] The inventor of the `null` reference, Tony Hoare, has said that they have "led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years" [66].

from within `a` and this makes it possible to *e.g.*, implement data structures that require multiple references to the same object. The owner (context) is indicated as a parameter at object creation, which means that the new object may be owned by some other context than the one where it was created. This makes ownership types even more flexible, while still powerful enough to protect an object's internal representation from being aliased through external references without requiring that the reference to the representation object is unique. The contexts as parameters to allow the programmer to create *e.g.*, container objects to hold objects that belong to an external context. An example often used to illustrate this is a linked list, where the node objects that link the list together are part of the linked list object's representation, while the objects stored in the nodes are owned by an external context [48]. This means that the stored data can be accessed from the external context, but the node objects are not available from there.

The many solutions suggested in the field and their difference in terminology, has made it difficult to *e.g.*, compare different solutions. Capabilities were introduced as an effort to collect the advances in the field under a unified terminology [25].

Capabilities

A capability, as described by Boyland et al. [25], is a combination of a memory address (a pointer or reference) and some access rights expressed through through the type. The access rights describe what operations the holder is allowed to perform on the object pointed out by the reference (*e.g.*, read, write, check identity) and also what the holder is allowed to do with the reference *e.g.*, sharing it by copying.

Pony [16] is an object-oriented language supporting concurrency by the use of actors [68] and a capability-based based type system to prevent data-races. The reference capabilities are used to express what possible aliases to the same object that are denied. The set of access rights is extended to six different capabilities for more fine-grained control:

1. Isolated (`iso`): Is mutable but denies both local read-write aliases and global read-write aliases. This means that there can only be one reference that can be used for reading and writing.

2. Transition (`trn`): Is mutable but denies local write aliases and global read-write aliases. This means that an object referred to by a `trn` reference capability can be written to from one single reference, but can be read from many aliases.

3. Reference (`ref`): Is mutable, but cannot be shared. Local read and write aliases are allowed.

4. Value (`val`): Is immutable and can be shared. Local and global read aliases are allowed.

5. Box (`box`): Is immutable but allows *either* local write aliases or local and global read aliases.

6. Tag (`tag`): Is opaque, that is no reading or writing is possible, but is possible to use to compare identities. Allows all local and global aliases.

Kappa is a capability-based type-system ensuring data-race freedom for object-oriented, concurrent programs [14]. A Kappa capability consists of a type and a mode. The type part defines the available interface of the object (or partial object) referred to. The modes are used to control how the capabilities may be aliased and shared between threads. *Kappa* defines six different modes:

- `linear` mode means that the resource can never be aliased.
- `thread` mode means that the resource may only be shared within the same thread.
- `locked` mode means that the resource may be shared, but operations on the resource will be guarded by locks.
- `read` mode means that the resource may only be read, not written.
- `subordinate` mode means that the resource is part of a surrounding object's private representation.
- `unsafe` mode means that the resource may be shared and must be used with some explicit concurrency control, *e.g.*, locks, to be used safely.

Accesses through a capability will always be mutually exclusive, that is while one thread is accessing an object through the capability, it cannot be updated by another thread.

Object in *Kappa* are constructed by assembling *traits* [69], which makes it possible to grant partial access to objects.

3.1.4 Dynamically Typed Languages

The actual data stored in memory will always have an intended use, a type, but the responsibility for keeping track of types and when the types are checked differs between different types of programming languages. In dynamically

typed languages, (just as in statically typed languages that use type inference) the programmer is not required to annotate the program with types. Unlike the type-inferred languages, dynamically typed languages perform no static checks or static analyses of types. Languages of this kind (*e.g.*, Lisp [34], Smalltalk [70], Perl [71], Python [72], Javascript [73]) have been and are still used alongside the statically typed ones. The first language in this category was Lisp [34], contemporary with early statically typed languages Fortran [33] and both are still in use today.

Dynamically typed languages allows the programmer to operate more freely and take the responsibility that the program will not encounter any errors when running and the conservative nature of type systems [45] has motivated the continued use and development of dynamic languages. Programs written in a dynamically typed language may be written in a more concise way, with less code.

```
1  class A:
2      def bar(self):
3          print("bar")
4
5  a = A()
6
7  c, d = 3, 3
8
9  if c > d:
10     a.foo()
11 else:
12     a.bar()
```

Listing 3.4: Code example showing a program which would be rejected by a conservative type checker since the method `foo()`, which is called on line 10 can not be found in an object of type `A`.

In the code in Listing 3.4, a class `A` is defined containing a method `bar`. A new object of the class `A` is saved in the variable `a` and in the `if`-statement at the bottom two method calls are made to `a`, the first one (which will never be executed) to a non-existent method `foo`, and second to the method `bar`. A static type checker would detect the absence of the method `foo` and reject the program, even though a reader of the code can easily confirm that it will run without errors.

Even though dynamically typed languages do not associate variable names with types and omit checking types before the program is allowed to run, there are types. All values have a type, but all type-related checks are deferred until

run time. In the program in Listing 3.4, the variable `a` is not associated with any type information, while the object pointed to by `a` will have the type `A`. No static checks are made to see if the methods called on the variable `a` exist, since it could refer to any kind of object, but the run time environment will perform a search for the methods actually called (`bar()`) starting in the object pointed to by `a`. Since variables are not associated with any static type, there are no mechanisms present that limit polymorphism *e.g.*, for method calls or return values.

In the code example in Listing 3.5, we see a method call to the method `o.getValue()` on line 14, and the type of `o` is only known when the call is executed. The `for`-loop on line 13 iterates over a list created on line 11. The list contains a mix of objects of the classes `A`, `B` and `C`. Since the variable `o` on line 14 will refer to different kinds of objects, the result of the method call will depend on whether the type of the object is `A`, `B` or `C`. For the types `A` and `B`, the method call will result in return values of type `str` and `int`, respectively, and these values will in turn be printed to the terminal. In the case where the type is `C`, however, the method `getValue()` can not be found and the result of the call will be a type error and that the program is aborted.

```
1  import random
2  class A:
3      def getValue(self):
4          return "A value"
5  class B:
6      def getValue(self):
7          return random.randint(0,10)
8  class C:
9      pass
10
11 l = [A(),B(),A(),B(),C()]
12
13 for o in l:
14     print(o.getValue())
```

Listing 3.5: Code example showing the use of polymorphism in Python.

In *e.g.*, Python, Smalltalk and Ruby, the call to an unidentified method can be captured in the receiver class to trigger the generation of code defining that previously non-existent method, installing it on the fly and then calling it. The run time types can also be used in dynamic analyses that aim to improve run time performance, *e.g.*, polymorphic in-line caching [74; 75].

The absence of static type information and type checking makes it easy for

```
1  f = open("exec_code.txt", "r")
2  code = ""
3  for line in f:
4     code += line
5  exec(code)
6  p = Person("Guido", "January 31")
7  print(p.__class__)
8  print(vars(p))
9  print("What attribute should be removed?: ")
10 attribute = input()
11 delattr(p, attribute)
12 print(vars(p))
```

Listing 3.6: Code example showing the execution of Python code read from a text file and stored as a string.

```
1  class Person:
2     def __init__(self, name, birthday):
3       self.name = name
4       self.birthday = birthday
```

Listing 3.7: The code file read in Listing 3.6.

dynamically typed languages to be highly dynamic also in other ways. Many allow executing code that is provided as text strings, and these strings may be put together at run time to allow the code to depend on information known only at run time. In Python, several different constructs are available to support this. The program in Listing 3.6 starts by opening a file called `exec_code.txt` on line 1 (which is the file shown in Listing 3.7). On line 2-4, all lines in the file are read and appended to the string stored in the variable `code`. On line 5 the function `exec()` is called with the string `code` as an argument, which will lead to that all contents of the string will be executed as Python code in this context. That will result in the definition of a new defining a class `Person`, shown in Listing 3.7. This class has an `__init__` method that sets two attributes in the new object; `name` and `birthday`. After the call to `exec`, on line 6, a new object of the class `Person` is created with the name "Guido" and the birthday "January 31" and a reference to the new object is stored in the variable `p`. On line 7, the result of looking up the `__class__` attribute is printed, which should show a string representation of `p`'s class. On line 12, all attributes of the object stored in `p` are printed to the terminal. Listing 3.8, contains the result of running the program from Listing 3.6. An

```
1 > python3 Code/example0.py
2  <class '__main__.Person'>
3  {'name': 'Guido', 'birthday': '31 January'}
4  What attribute should be removed?: birthday
5  {'name': 'Guido'}
```

Listing 3.8: Running the code from Listing 3.6.

object was created and on line 2 we see its string representation containing the class name `Person`. On line 3, all attributes of the object are printed.

Many dynamically typed languages also allow *e.g.*, addition and removal of attributes and methods at run time. The program in Listing 3.7 prints all attributes in the object referred to by `p` on line 8. On line 9 the user is asked what attribute should be removed and the name of that attribute is saved in the variable `attribute`. On line 11, the attribute with the name stored in the variable `attribute` is removed from the object `p`. On line 12, all attributes of the object are printed to the terminal again. The result of running this code is shown at the end of Listing 3.8. On line 3, all the attributes and their values are printed out on the terminal. On line 4, the user answers "birthday" to the question of what attribute should be removed and when the object's attributes are shown again on line 5, the attribute `birthday` (not only its value) is no longer there.

The examples of highly dynamic behaviour described above necessitates run time checks for for example the presence of a method in an object every time that method is called, even if the method call was successfully executed earlier with the same receiver variable. The object stored may have been exchanged with an object of another type or the object itself may have been modified.

The use of dynamicity in terms of using the same variable to store values of different types at different times, to use dynamic features to add or remove attributes in objects, or dynamic code evaluation to execute code from strings that may be assembled at run time is always possible and could be used in ways that obfuscate the code and slow down execution. In the same way as omitting type checks transfer the responsibility of keeping track of types to the programmer, the freedom to use dynamic features to make changes to the running program requires the programmer to be restrictive and use it only in situations where it is assessed necessary.

3.1.5 Mixing Dynamic and Static Types

Commonly, languages are either statically typed or dynamically typed and both approaches come with advantages and disadvantages.

An advantage of statically typed languages is that they provide the programmer with more support and enable more optimisation resulting in faster programs. A disadvantage is that programs will be rejected and not allowed to run if the static type information (by annotation or inference) is not complete or contains errors, even if that part of the program will never be executed. They will also reject programs that cannot be given a static type, *e.g.*, the program in Listing 3.6.

An advantage with dynamically typed languages is that they allow fast and experimental development. A disadvantage is however that if the type of a value does not match what is needed at some point in the program (*e.g.*, a method called is not available in the receiver object), the program execution will be aborted.

The conservative nature of static type systems and the lack of support given by dynamically typed languages has motivated a variety of compromises to give access to advantages from both approaches in the same language. There are gradual types [76] (static types in dynamically typed languages) in *e.g.*, Hack [77], TypeScript [78], mypy [79], and Julia [80]. There are also dynamic types in statically typed languages [81] present in *e.g.*, in C# [42] and Rust [82].

The basic idea of gradual typing [76] is that if the programmer chooses to leave out type annotations in the code, the program can be run instantly as if written in a dynamically typed language, but type checks will be made at run time. If the programmer instead provides type annotations, the program (part) will be considered to be statically typed, and the types will be checked at compile time. If a program consists of parts that are statically typed and parts that are dynamically typed, run time type-checking will be performed at the boundaries between the parts and omitted within the statically typed code that has been checked at compile time.

Since Python 3.5 [83] it has been possible to use type hints to annotate Python code with type information, which was accepted in the form presented in PEP 484 [84] after many years of discussion. These type annotations are entirely optional and the type information is not used at run time, although the implementation allows third-party libraries to implement this. The motivation behind these type annotations is first of all to improve the possibilities to use static analyses and refactoring, but also to enable run time analysis and allow run time code generation based on type information [84]. In addition to this, external static type checkers are available *e.g.*, mypy [79], Pyre [85],

Pyright [86], that can be used to perform static type checking on Python programs containing type annotations.

Typescript [78] is a gradually typed language that extends JavaScript with static type checking using a combination of type inference and explicit annotations. This means that all JavaScript programs are also valid TypeScript programs. Values may be explicitly omitted from the static type checking by use of the type `any`, which will defer checks until run time [87]. The programmer can define typed interfaces for composite object types which will be type checked when used. Types can be parameterised with other types, which means it is possible to specify the content type of collections (*e.g.*, specify that an array will only contain objects of some specific kind). Type checking is structural. After the type checking is done, the compiler removes the type annotations from the type-checked program, which results in a valid JavaScript program.

In the dynamically typed language Julia [80], the programmer may annotate expressions and variables with type information to increase readability, discover errors and to underpin the language's support for multiple dispatch. Dispatch is the mechanism used to select which method to run given a method call. In Julia, the dispatch uses the number of arguments together with the type of all arguments to perform this selection. This is unlike most other object-oriented languages which usually only uses its first "argument" (the receiver) for method dispatch. In Julia, all types are object types, types can be parameterised using other types, and the relations between types are defined nominally (a subtype uses the name of its supertype to declare the relation).

The type annotations do however not change the fact that Julia is a dynamically typed language where only values have types. The provided type information will only be used at run time to assert at that the value of expressions and variables have the expected type. It makes multiple dispatch method calls possible, and it will also provide information to the compiler to enable optimisations.

The language Hack [77], which is a dialect of PHP developed by Meta, supports gradual typing. Most PHP programs are valid Hack programs, but programs using deprecated functionality and a few features that were decided to be incompatible with static typing (*e.g.*, the `extract` function which inserts variables to the current context from an array). Statically typed Hack programs are type checked statically. Code can however be left without type annotations or annotated with type information at different levels: "soft type hints" or strict types. If the program fails to follow the strict type annotations in the code, which can happen in the borderlands between statically typed and dynamically typed code, the result will be an error at run time. If the type instead was provided as a soft type hint, the program will output a warning (also

at run time). This is convenient *e.g.*, for dealing with transferring large code bases from plain dynamically typed PHP to fully annotated Hack.

Hack's type system is nominal and new types may be created using type parameters. Type inference is used for some local variables, and if no type can be decided the type inferred will be `any`.

All the examples above are dynamically typed languages where type annotations have been introduced that will be checked in different ways. There are also examples of the opposite, that is statically typed languages which have introduced the possibility to use dynamic types in some situations.

Rust [82] is a statically typed language with a focus on memory safety and performance. It however also includes the possibility for the programmer to shut down some of the help provided from the language, take over the responsibility to make sure that the program will not for example dereference null pointers, by using the `unsafe` mode. In addition to this, there is also a type `Any`, that makes it possible to *e.g.*, create collections that can store values of different unrelated types or write functions that will accept arguments of any type.

In C#, the programmer can annotate variables to defer type checking until run time [88]. The type `dynamic` will store values of any type and values stored in a variable of the type `dynamic` will be accepted by the type checking in any typing context. At run time, the type `dynamic` will be replaced by the type `Object`, but *e.g.*, method calls will be accepted also for methods that are not present in `Object`. A variable declared as `dynamic` will also be implicitly converted to any type to fit into the program context but if the dynamic value fails to fulfill the requirements of its context at run time, the program will fail.

3.1.6 Typability

The term typability is often used in the context of programming languages using type inference. In this context typability is the ability of the analysis to infer the types [89], but the term is useful and relevant also in other contexts.

Programs have dynamic behaviours that can be observed at run time, but using *e.g.*, types, we can also reason about some behaviours without running the program. Following this line of reasoning, typability is a question about what kinds of behaviours can be identified using different kinds of types.

Typability in Dynamically Typed Languages

In dynamically typed languages, typability can be interpreted as *e.g.*,:

- The *stability* of the types, that is how common it is that the operations

available in a type changes. For a class-based object-oriented language the stability of types would translate to the stability of its classes, and the objects' adherence to their classes.

- The *regularity* in use of the available types, that is how common it is that variables refer to values of the same or in some way related types (related by subtype polymorphism, or structural similarities).
- The possibility to retrofit programs (completely or partially) with static types, that is the possibility to use type inference.

In the work presented in this dissertation, Paper I studies the stability of types in Python programs and Paper II studies the regularity of types in Python programs. The results from both Paper I and Paper II are useful for understanding the challenges met in retrofitting programs written in dynamic languages with static types.

Typability in Statically Typed Languages

In statically typed languages, static type checking is used to support the programmer in many ways, but the information carried by the type is often limited to its available fields and methods and relation to other types. Typability in this context can be interpreted as the possibility to use more specialised types that contain more information provide more help to the programmer, *e.g.*, for managing aliases as discussed in Section 3.1.3.

Paper III, included in this dissertation presents array capabilities, which is an extension of reference capabilities for supporting programmers in writing code operating on arrays. Typability in this context is that the specialised types allows the programmer to express dynamic behaviour in terms of distributing access to array parts. This makes it possible to detect programs containing data-races, since those programs cannot be typed, while a program without data-races may still be rejected. All programs that are typeable are however guaranteed to lack data-races. Array capabilities thus provide support for the programmer in implementing parallel array algorithms that will never result in data-races.

3.2 Arrays

This chapter provides some necessary background about arrays for the second theme of this dissertation "Improving support for programming with arrays".

An array is a composite data type with the capacity to store a *fixed number* of values (*elements*) of the same type and size and where all the elements are

stored physically in sequence in the computer's storage. Access to elements is achieved by subscripting the array with an *index*, which is usually a number. Any element number n in the element sequence will be stored at an offset n times the storage size of the elements from the start of the memory location allocated for the array.

The array is a conceptually minimalistic data structure available in most programming languages, both in older, all-purpose and more low-level languages like C [90] and newer, more high-level languages like Julia [91]. Even Konrad Zuse's programming language Plankalkül that was defined in the 1940s contained arrays [92].

Arrays are stored contiguously in memory resulting in low memory overhead and fast access time compared to other data structures. Arrays are often used to implement other more advanced data structures like maps (*e.g.*, HashMaps in Java, or Dicts in Python). At the surface, arrays are similar to maps in that they can be understood as mapping an index to a stored element, but there are large differences in what goes on behind the scene. Bare-bones arrays, as arrays in assembly languages are not as much an abstraction as they are a consequence of how computer memory works. Knowledge about (a pointer to) the location in memory where our data is stored combined with knowledge about the amount of memory taken up by each element can be used to retrieve the *nth* element since the elements are stored in sequence. To find the *nth* element, we simply move the pointer forward n times the size of the element (given that the first element is regarded as number 0). An array-based map, on the other hand will use the its key value to calculate an index, which will be used to look up the element in the underlying array. The calculated index will however not come with any guarantees that it is unique, which means that a map needs to implement some collision handling and search for the element. Both arrays and maps will in general provide constant time access to their elements, but the map will always have to perform additional work in calculating the index. Iterating over all elements of an array will be somewhat faster than iterating over all elements of a probing map since the map will not use all slots. A map implemented using separate chaining would need perfect distribution to match the array iteration.

Although the array is a minimalistic data structure even in high level languages, there is still room for some variation between implementations made for different languages. Some of the differences can be found in the attributes of the array, like if the array is aware of its length or not.

Most languages provide several ways of creating new arrays; literals, empty arrays with a certain length (the capacity to store a certain number of elements), list comprehensions [32] etc. Since different ways to create arrays in the same language in general produces the same kind of array as a result (same struc-

ture, same data) these differences are of little interest in the context of this dissertation.

Most arrays use integer numbers to index the elements (typically starting with index 0), but some languages allow the use of *e.g.*, characters or any range of numbers instead (*e.g.*, Pascal [93]). Tuples can be used as indices for accessing elements in multi-dimensional arrays as in Julia [91], where `(1,1)` would mean the element in row 1 (the first row) and column 1 (the first column) in a two-dimensional array. PHP allows using any type as index for its arrays, but these arrays are in fact implemented as ordered maps [94] which are outside of the scope in this context. It is common that the numbers used to refer to the elements start at 0 (C, Java, C#, Go, Python, etc.), but some languages store the first element at position 1 (Julia, Lua, MATLAB, R, etc.).

Other differences are more part of the array data structure interface, like slicing (further discussed in Section 3.2.3) and the use of iterators.

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[]) {
4    int arr[] = {11,33,55};
5    int initially_empty_arr[3];
6    initially_empty_arr[0] = 22;
7    initially_empty_arr[1] = 44;
8    initially_empty_arr[2] = 66;
9    printf("%lu\n", sizeof(arr)); // three ints : 3 * 4 = 12
10   printf("%lu\n", sizeof(arr)/sizeof(int)); // 12 / 4 = 3
11   int *p = arr;
12   printf("%i\n", arr[1]); // 33
13   printf("%i\n", *(p + 1)); // 33
14   return 0;
15 }
```

Listing 3.9: Code example showing some declarations of arrays in C.

3.2.1 Low-Level Arrays

C In C [90] the array does not fulfil the requirements we have for an abstract data type [38], but resembles arrays in assembly languages.

If we, like on line 4 in Listing 3.9, declare an array `int arr [] = {1,3,5}`, memory will be allocated for three `int` values in sequence and the values provided will be stored there. It is also possible to declare an array with a given size, as is shown on line 5 in the same Listing. Even though the size is part of the type of `initially_empty_arr`, it is of little use. The

size of the array cannot be changed after creation in either of the cases and the array does not keep track of its length, but the compiler keeps track of its size in memory, as is shown on line 9 in Listing 3.9. This line will print 12 to the terminal (the size of three ints). To find out the length, the programmer has to manually divide the size of the whole array with the size of the elements, which is shown on line 10 in Listing 3.9. The information about the size of the elements is however only available in the scope where the array was created.

On line 11, the `int` pointer `p` is set to point to `arr` (which actually means that it points to the first element in `arr`). The subscripting used to access a specific element, *e.g.*,`arr[n]` to access the element number `n + 1` in the array (shown on line 12) `arr`, is equivalent to `*(p + n)`, that is moving the pointer `n` steps (the size of one element) forward and reading the value found at that location (shown on line 13. Both line 12 and line 13 will print out the same element from `arr`, that is 33.

This array implementation has no mechanisms to prevent reading its elements inaccurately. The array stores no metadata, it is not aware of its size or the size of its elements, so there is no bounds checking neither for the array as a whole or for the bounds between the elements stored. For an array storing elements of a size *s* it is possible to access the array contents even if we do not access the elements at even *s* size intervals from the storage's starting position.

Go and Rust The array type both in Go [95] and Rust [82] is a fixed-size, low-level construct that has no other "fields" than its elements and it provides only basic operations in itself. Subroutines are available to find out the length of the array. Both languages use bounds checking at run time to secure that indexing is within the array's bounds.

In both Rust and Go, (unlike in C) the array can *e.g.*, be passed to a subroutine using value semantics. This may result in time and memory consuming copying although the compiler may use optimisation techniques to avoid this. A pointer to the array can be passed instead, but that has to be done explicitly by the programmer.

There are other and more flexible types available *e.g.*, `slice` in Go, which is commonly used instead of the fixed-size array [96]. In Rust there is *e.g.*, a `slice` and a `Vector` type. The `slice` types are discussed further in Section 3.2.3.

C++ The array implementation in C++ was equal to the one in C until the release of C++11, when the `array` class was introduced [97; 98]. Instances of this class are aware of their size (through a template parameter that is set at compile time) and provide an extended set of useful operations at a higher level of abstraction. Two example operations are the `front()` and `back()`

methods that returns the first and last element of the array respectively. These operations both operate at a higher level of abstraction than mere array subscripting like `arr[0]`, where the programmer does not need to translate the "first" to index 0, or manually keep track of the length of the array, subtract 1 and look up that index to retrieve the "last". The interface also supports the creation of several different kinds of iterators to iterate over all elements of the array *e.g.*, from start to end and the other way around. The arrays created using the array class are still fixed in size after creation, elements all have the same type and they are stored in an unbroken sequence in memory. Indexing is also still 0-based. It is, however, still possible to acquire a pointer to the first element of the array by calling the method `first()`. This pointer can then be used in the same way as in C to access other elements using pointer arithmetic.

Similar to arrays in Rust and Go, C++ arrays can be passed to subroutines using either value or reference semantics.

In the end, this means that although the array class provides an abstract data type with a defined behaviour like iterators, the pointers give programmers freedom to access the array elements in any way they like.

3.2.2 Higher-Level Arrays

Java Java's arrays are declared with a static type for all elements. The dynamic type (the actual type of the elements) may however be a subtype of the declared type, as with other Java type declarations [99]. The array's length is not part of the array type: all arrays of *e.g.*, the type `int[]` or the type `String[]` have the same type regardless of their length. Once the array object has been created its length cannot be changed and the length value is saved as a final (immutable) instance variable in the array object.

Although the array is not a class, all arrays are objects and they hold a number of array-specific methods and methods inherited from the `Object` class. All array instances storing the same element type share a common anonymous class object.

All accesses in Java arrays are bounds checked at run time and all array assignments are also type checked at run time, which is necessary since array covariance is allowed. The array object wraps the actual array and it is not possible to *e.g.*, create a pointer to a certain position in the array directly.

C# Similar to arrays in Java, C# arrays are objects. The C# array type contains information about its number of dimensions (dimension 1 is a one-dimensional array like in the other examples above) and the length of each dimension is set when the array object is created. Similarities between C#'s and Java's arrays are that all elements in an array have the same common super-

type, that the size in number of elements cannot change after creation and that enumeration starts from 0. In C# the array is a class and the `System.Array` class is the abstract superclass for all array classes. All arrays in C# implement the `System.Collections.Generic.IList<T>` interface. This means that arrays can be used interchangeably with any collection. This leads to an array construct at a higher level of abstraction than in Java, but the data structure is still a true array in that it stores its values (primitive values or the references to objects) in a contiguous block of memory.

Python Python comes in many implementations, and what is considered here is the reference implementation, known as CPython [100]. In Python, the data type used for sequential storage is called `list`, which in reality is an array implemented to have dynamic size, not a linked list. Since all values in Python are objects, the `list` will contain references and there are no restrictions on the type of the objects stored in the `list`.

Some illustrating code can be found in Listing 3.10, where an empty `list` is created on line 1. On line 2 the list is passed as an argument to the built-in function `len()`, which returns the length of `l`, in this case 0. The length of a `list` can be increased or decreased at run time using the methods `append()` and `pop()` to add or remove elements respectively. An element at a specific index can be removed without leaving "holes", and elements can be inserted between already existing elements. On line 3-4 in Listing 3.10, a `for`-loop populates the list with the values 0-5. On line 5, the length is checked again using the `len()` function, which now returns 6. The remaining code in Listing 3.10 will be discussed in Section 3.2.3.

A `list` is a dynamic array of references to objects. The `list` is always allocated to have room for more elements than needed. The list object itself stores a reference to the underlying array, the current number of elements, and the number of elements there is room for [83]. When elements are appended, they are inserted in this extra space. If many elements are inserted, and the array runs out of extra space, a reallocation (including copying of all existing elements) will take place, but look-up will always be performed in constant time.

Python arrays are bounds checked at run time and accesses outside the current array's bounds will result in an exception. The Python list and slicing is discussed further in Section 3.2.3.

3.2.3 Slicing and Array Partitioning

A *slice* is a section of an array. It can be created either by actual copying of the selected section of the original array or it can be a "masked view" into

```
1 l = []
2 length = len(l) # length will be set to 0
3 for i in range(0,6): # produces the list [0, 1, 2, 3, 4, 5]
4   l.append(i)
5 length = len(l) # length will be set to 6
6 s1 = l[:4] # s1 is [0, 1, 2, 3]
7 s2 = l[2:] # s2 is [2, 3, 4, 5]
8 s1[2], s1[3] = s1[3], s1[2] # s1 is [0, 1, 3, 2], s2 is [2, 3, 4, 5]
```

Listing 3.10: Code example showing the use of slicing on Python lists.

the original array. In the example languages discussed below, both of these approaches are accounted for.

Python The Python `list` supports slicing, that is creating sublists of the original `list`. A sublist created by slicing in Python will be a new `list` containing a subset of the elements from the original `list`. Since all elements are objects, the references will be copied from the original `list`.

We return to the code in Listing 3.10, where a `list l` is created and populated on lines 1-4. On lines 6 and 7 two new `lists` are created, `s1` and `s2`, where `s1` will contain the elements from the original `list`'s index 0-3 and `s2` will contain the elements from the original `list`'s index 2-5. The new `lists` (the slices) are however not referring in any way to the original `list` or each other.

Both `s1` and `s2` contain the elements from the indices 2 and 3 and in a case where the elements are objects with mutable state (which is not the case with these integers) this makes it possible to access and update the actual elements independently from both `s1` and `s2`. Since the slicing may lead to aliasing, updates to an element from one of the slices may lead to unexpected behaviour when the element is accessed from the other slice. Any reassignment or reordering of the indices in `s1` will not have any impact on `s2` and the other way around.

Implementing *e.g.*, quicksort using slices in Python would require merging the slices in the end and the sorting will not be done in-place in the original list.

Go The slice type of Go has a dynamic size and does not store any elements, but instead gives a view of a subset of the elements of an array. A slice has a certain length, and it also has a capacity that corresponds to the length of the underlying array, counted from the position of the first element of the slice.

```
1 fib := [6]int{0, 1, 1, 2, 3, 5} // int array of size 6
2 odds := [4]int{1, 3, 5, 7} // int array of size 4
3 evens := []int{2, 4, 6, 8} // literal int slice (no size given)
4 var s1 []int = odds[:2] // s1 is [1 3]
5 var s2 []int = odds[2:] // s2 is [5 7]
6 s1 = s1[:3] // s1 is now [1 3 5]
7 s2 = s2[:cap(s2) - 1] // s2 is now [5]
8 s1[1], s1[2] = s1[2], s1[1] // s1 is [1 5 3] and s2 is [3]
```

Listing 3.11: Code example showing the use of arrays and slices in Go.

The slice provides a view of an underlying array and this view may be of the same length or shorter than the length from the slice's start index in the underlying array to the end of the underlying array (the slice's capacity). The slice type length is flexible (it can be increased or decreased) and not part of the type. The slice's length can be increased to incorporate more of the underlying array, that is any elements in positions after the view given by the slice. To find out the length of arrays (and slices), Go provides the built-in function `len()`. To find out the capacity (that is the maximum length) of slices, Go provides the built-in function `cap()`.

Listing 3.11 contains code that shows examples of arrays and slices in Go. On line 1, an `int` array of size 6 is created and on line 2 an `int` array of size 4 is created. These arrays do not have the same length and therefore they do not have the same type, which means that we *e.g.*, cannot assign the variable `fib` with the array stored in the variable `odds`.

On line 4 and 5 two slices are created from the array `odds` and the important difference between these slices and the arrays created above is that the type does not contain any information about the slices' sizes. The length of both `s1` and `s2` is 2, but their capacities (that is the remaining length of the underlying array) differ. The capacity of `s1` is 4, since the array contains 2 positions after the "location" of `s1`. The capacity of `s2`, however is 2, the same as the length, since the last element of `s2` is also the last element of the underlying array.

This means that the slice `s1` can be extended (but not `s2`), and on line 6 `s1` is extended to the the length 3. On line 7 the length of `s2` is decreased to 1. Finally, on line 8, the value in at index 1 and 2 in `s1` are swapped with each other. This means that `s1` will end up with the values `[1,5,3]` and it also changes `s2`, which now contains the value `[3]`. Also the original array has changed to `[1,5,3,7]`.

```
1 fn main() {
2     let a1 = [11, 33, 55, 77, 99];
3     let mut a2 = [0, 22, 44, 66, 88];
4
5     let s1 = &a1[..3]; // [11, 33, 55]
6     let s2 = &a1[2..]; // [55, 77, 99]
7     let s3 = &mut a2[2..3]; // [44]
8     s3[0] = 77; // a2 is now [0, 22, 77, 66, 88]
9 }
```

Listing 3.12: Code example showing the use of arrays and slices in Rust.

Rust As in Go, the Rust slice is a view into an existing array. We may create several slices that overlap from the same array, but a difference from Go is unless the slices are immutable, only one of these overlapping Rust slices may be in use at the same time. Rust uses ownership and borrowing to control this. An example with some slices is shown in Listing 3.12. On line 2-3, two arrays are created, `a1` is immutable and `a2` is mutable. On line 5-6, two immutable slices are created from `a1`. They can be used for reading, but the values in the slices cannot be changed. On line 7, a mutable slice is created from the array `a2`. On line 8, index 0 in the mutable slice `s3` is set to 77. This means that the slice `s3` now contains `[77]` and the array `a2` is now `[0,22,77,66,88]`.

Slices in Relation to Array Capabilities Paper III, presents array capabilities that define a set of operations to split and merge arrays. The array capabilities are, like slices in Go and Rust, a view into an underlying array. Modifications of the underlying array are allowed through an array capability, and more than one array capability can provide views into the same underlying array. The array capabilities provide a set of more flexible splitting operations than Rust and still come with a guarantee that no overlaps exist between mutable array capabilities. This results in that changes made will only affect the modified array capability and the underlying array, never any of the other existing array capabilities. Array capabilities that are immutable may however contain overlapping elements.

4. Summary of Papers

Having given necessary background on types and arrays, this chapter presents the contributions made in this dissertation based on the research questions presented in Section 1.1.

4.1 Understanding how types are used in programs written in dynamically typed languages

To answer the research questions from Theme I, I have performed two studies presented in Paper I and Paper II. At the time when the studies were made, there was a large interest in developing type systems for developing type systems for dynamically typed languages [1–9]. The absence of a conservative static type system allows the programmer to freely assign values of different types to the same variable at different times during program execution. This, combined with the fact that dynamic languages commonly provide powerful tools for *e.g.*, reflection (including adding and removing *e.g.*, attributes at run time) and metaprogramming *could* result in programs that would not be accepted by a static type system. The fact that programs are written *e.g.*, in Python may indicate that programmers expect to gain advantages from using these features. If, however, variables are always used to store objects of the same type, and if dynamic features that make types unstable are seldom used in practice, it may be possible to *e.g.*, retro-fit programs written in dynamic languages with a static type system.

Paper I present results from examining the occurrences of dynamic behaviour and Paper II presents results from studying the use of polymorphism, both studies are made on real-world programs showing that retro-fitting static type-systems to entire programs would not be possible.

4.1.1 Paper I: Dynamic Behaviour

The instability of types (that is that the set of available methods and instance variables changes during run time) caused by use of dynamic features, such as reflection and run time code generation, was studied in 19 open source Python programs. The features traced in this study are listed in Figure 4.1. The results

Categories			
Introspection	Object Changes	Code Generation	Library Loading
`hasattr`	`del` *attribute*	`eval`	`_import_`
`getattr`	`delattr`	`exec`	`reload`
`_getattr_`	`_delattr_`	`execfile`	
`_getattribute_`	`setattr`		
`vars`	`_setattr_`		

Figure 4.1: Dynamic Python features traced in the study presented in Paper I, separated into the categories introspection, object changes, code generation and library loading.

show that dynamic features are used in real-world Python programs resulting in programs that would be hard to type.

All programs in the study used many (between 5 and 9 out of 15) of the features traced, and all programs used features from all categories.

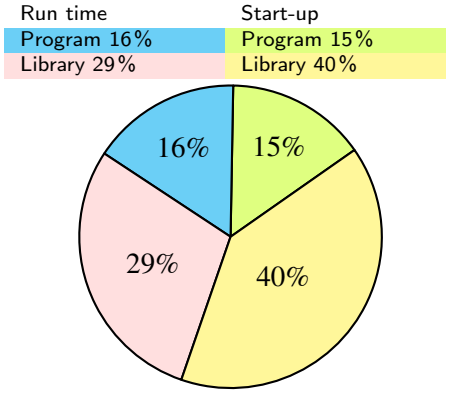


Figure 4.2: Distribution of dynamic features over libraries and program-specific code during start-up and run time.

Figure 4.2 shows that dynamic features are used both in library code and in application-specific code, during the start-up phase as well as the run-time phase. The start-up phase covers the loading of the graphical user-interface and ends when the execution of the main loop of the program starts. Dynamic feature use is only slightly more common in the start-up phase than in the run time phase (55 % of the dynamic features were found during start-up and 45 % during run time). The use of dynamic features is higher in library code than in program specific code (69 % in library code and 31 % in program specific code), but dynamism is by no means encapsulated in libraries.

The results show that dynamic behaviour is neither buried in library code, nor predominantly occurs at program start-up time. This excludes solutions that would restrict the use of *e.g.*, reflection once programs stabilise and also

makes it impossible to easily identify the boundaries to make between typed and untyped modules.

If retrofitting type systems for Python code, measures must be taken to account for *e.g.*, widespread use of reflection that create attributes from string values (with the implications that a class' interface cannot be known at "compile time") to make the type system work with actual Python code.

4.1.2 Paper II: Polymorphism

The study of polymorphism was made on the traces from the 36 real-world open source Python programs. We instrumented the Python interpreter to output information every time a method call was made.

A *call-site* in a program is a point (on a line in a Python source file) where a method call is made. The call-site consists of two points where different types may emerge depending on the execution path: the receiver, and the arguments. This study focused on the receiver types. To a great extent, the call-sites in the 36 studied programs were either only logged once and these call-sites were therefore only logged with one single receiver type. We separated these *trivially monomorphic* call-sites from other call-sites which were logged more than once but still with only one receiver type, the *observably monomorphic*. The call-sites that were only logged once were not classified further and our trace-based method could not exclude the possibility that a different run of the same program might observe polymorphic behaviour for the same call-site. The share of monomorphic call-sites (including single call) ranged between 88-99 % with an average of 96 % (see Figure 4.3).

This suggests that large parts of the programs could be typed using simple and conservative nominal types. However, when looking in detail at the parts of the programs that were polymorphic, another story emerges. Figure 4.4 shows the maximal polymorphic degree for all runs of all programs, ranging

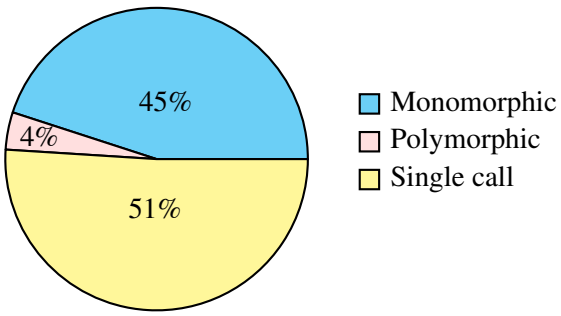


Figure 4.3: Distribution of call-sites between polymorphic, single call and monomorphic in all programs.

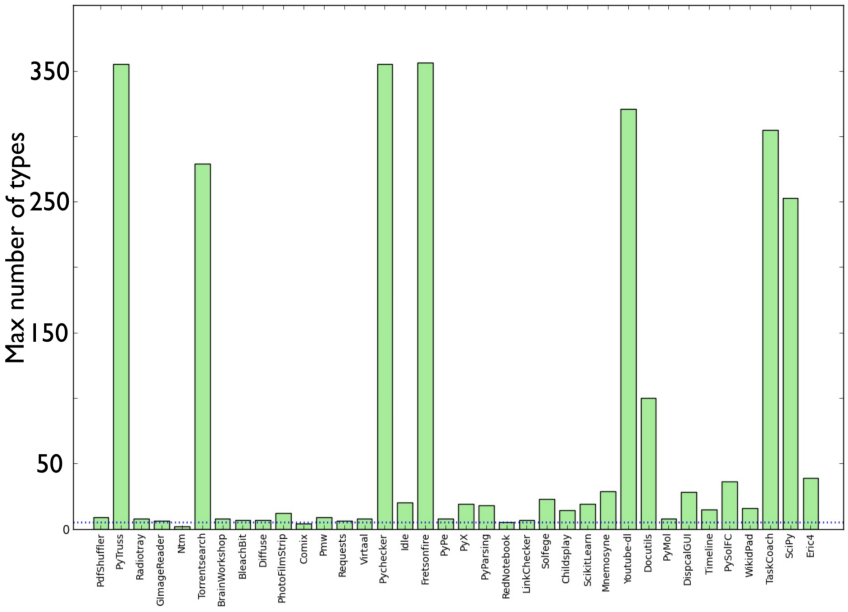


Figure 4.4: Max number of types seen at a call-site for the programs included in the study. The blue dotted line at the bottom marks the the border between polymorphism and megamorphism at 5 receiver types.

from 2 to 356 receiver types. The blue dotted line marks the border between *polymorphism* and *megamorphism* at 5 receiver types. Only 3 programs contain no megamorphic call-sites at all.

Figure 4.5 shows the degree of polymorphism for all polymorphic and megamorphic call-sites. Again, the blue dotted line marks the border between *polymorphism* and *megamorphism* at 5 receiver types. The vast majority, 88 %, of all polymorphic and megamorphic call-sites are not megamorphic. 78 % of the polymorphic call-sites had a polymorphic degree of 2, that is two different receiver types. While these numbers show that megamorphic call-sites are relatively rare, they are not concentrated to specific programs. Almost all programs (33 of 36) exhibited some form of megamorphic behaviour. In 30 out of 36 programs, 1 % or less of all call-sites were megamorphic, in both application code and library code.

We analysed the typability of individual call-sites using different typing approaches starting with nominal types and nominal types with parametric polymorphism.

In Figure 4.6 the bars represent the typable share of individual call-sites when using a nominal type system and in Figure 4.7 the staple represents the

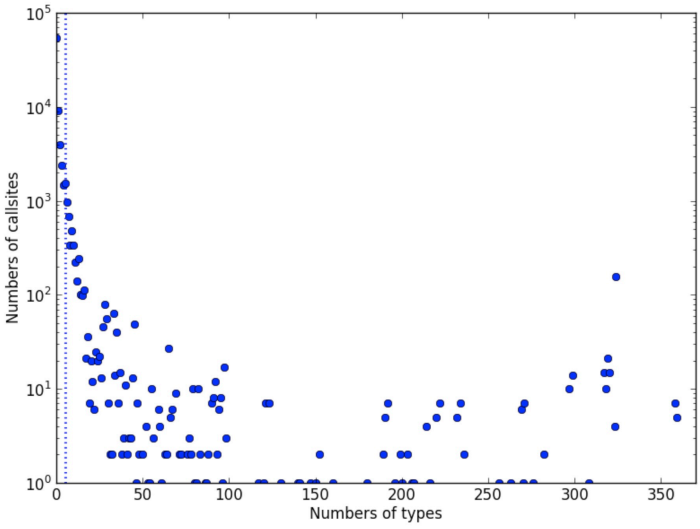


Figure 4.5: For all non-monomorphic call-sites, the plot shows the number of call-sites logged with a specific number of receiver types. The blue dotted line marks the border between polymorphism and megamorphism at 5 receiver types.

typable share of the individual call-sites when using a nominal type system with parametric polymorphism. In both figures, the green part shows the share of call-sites that were logged only once, the yellow part shows the share of observably monomorphic call-sites, and the red part shows the share of the call-sites that would be typable using a nominal and a nominal type system with parametric polymorphism, respectively.

The nominal type system could be used to type between 88.1 % and 99.9 % of the call-sites, with an average at 97.4 %.

A nominal type system with parametric polymorphism performed slightly better, it could be used to type between 88.9 % and 100 % of the call-sites, with an average at 97.8 %.

The analysis made on individual call-sites may however over estimate the typability *e.g.*, when the code contains value-based overloading as in Listing 4.1.

If the method `vbo` on line 13 in Listing 4.1 is always called using a first argument of either type `A` or `B` and a second argument which is always a boolean, always `True` when `a` is of type `A` and always `False` when `a` is of type `B`. When analysing the call-sites individually, the call-sites on lines 16 and 18 will always have the same receiver type and could be given a static type. In this case, however, the call-sites on line 14, 16 and 18 must be analysed together in a *cluster*.

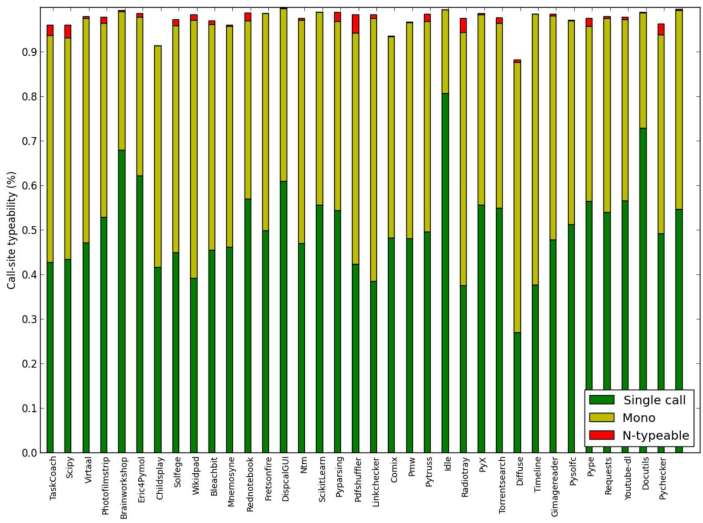


Figure 4.6: For each program the staple represents the typable share of the individual call-sites when using a nominal type-system. None of the programs were typable to 100 %, with an average at 97.4 %.

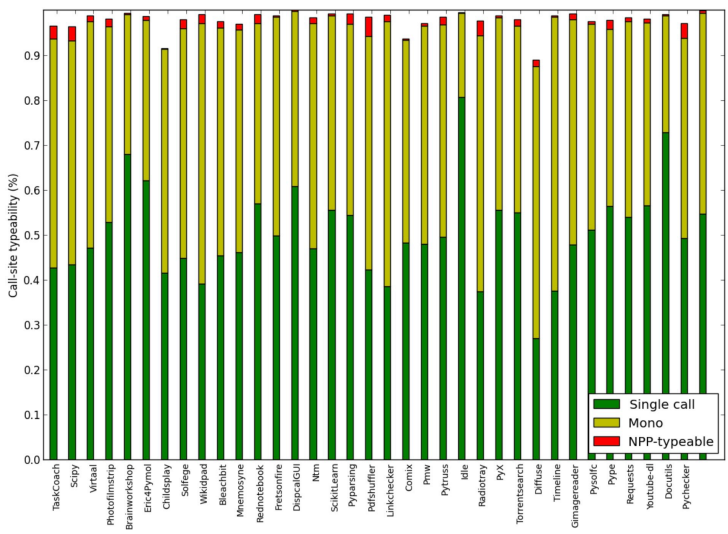


Figure 4.7: For each program the staples show the share of individual call-sites that would be typable using a nominal type system with parametric polymorphism, which in all cases could be used to type between 88.9 % and 100 % of the call-sites

```
class A:
   def foo():
      pass
   def bar():
      pass

class B:
   def foo():
      pass
   def baz():
      pass

def vbo(a, b):
   a.foo()
   if b:
      a.bar()
   else:
      a.baz()
```

Listing 4.1: Code example taken from Paper II showing the use of value-based overloading in Python.

Definition (taken from Paper II)

A cluster is a set of call-sites, from the same source file, connected by the receivers they have seen. For all pairs of call-sites `A` *and* `B` *in a cluster, they have either seen the same receiver or there exists a third call-site* `C` *that has seen the same receiver as both* `A` *and* `B`*.*

The typability analysis on clusters was made using a nominal and a structural approach. In Figures 4.8 and 4.9, the staples show the share of the call-sites that could be typed using nominal types and structural types respectively. As in for the individual call-sites, the green part shows the share of call-sites that were logged only once, the yellow part shows the share of observably monomorphic call-sites, and the red part shows the share of the call-sites that would be typable using a nominal type system without parametric polymorphism and a structural type system, respectively.

The nominal type system could be used to type between 91.9 % and 97.8 % of the clusters of call-sites (95.6 % on average).

The structural type system performed slightly better and could be used to type between 94.8 % and 98.4 % of the call-site clusters (96.7 %, on average).

In conclusion, for the programs included in our study:

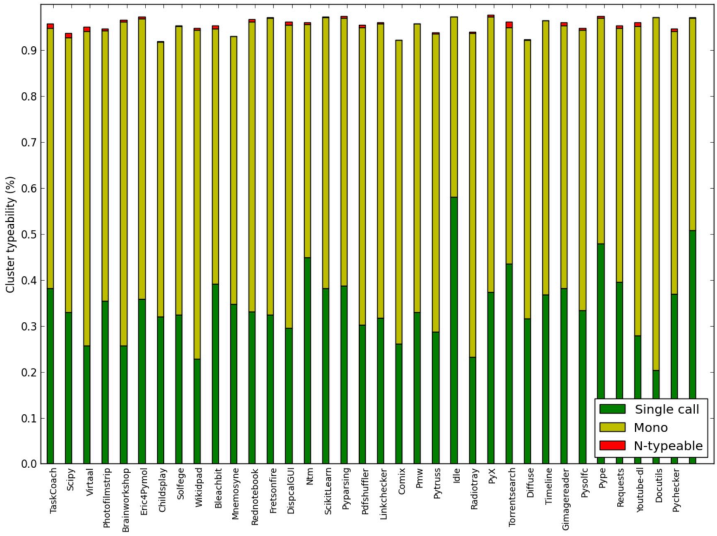


Figure 4.8: For each program the staples show the share of clusters of call-sites that would be typable using a nominal type system.

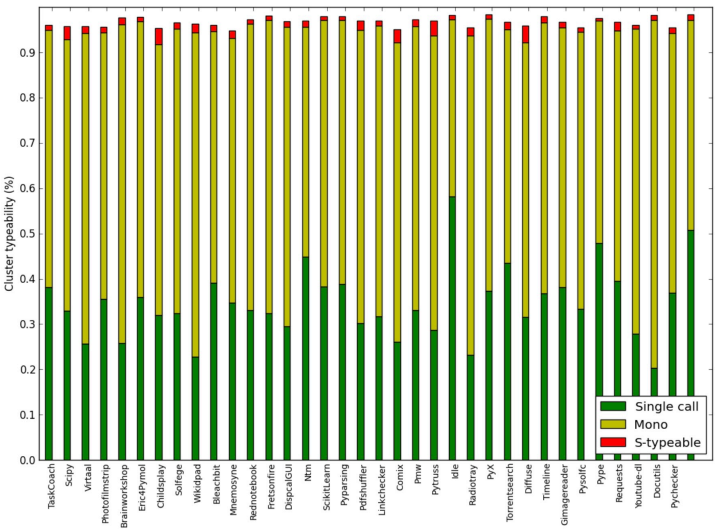


Figure 4.9: For each program the staples show the share of clusters of call-sites that would be typable using a structural type system.

- most call-sites are not polymorphic or megamorphic, but when they are, our simple nominal types cannot in general be used to type them. By extending the nominal type system with parametric polymorphism, we can type more call-sites for all programs, but the nominal types are not powerful enough to type whole programs even when extended with parametric polymorphism.

- clusters are predominately monomorphic or single call. Structural typing performs slightly better than nominal typing in typing call-site clusters, but still cannot in general type whole programs

  Nominal and structural typing is not a deciding factor when adding static types to programs written in dynamic languages, but more powerful type systems would be needed.

4.1.3 Notes on the Selection of Programs for Papers I-II

In Paper I, the study was performed on 19 open source Python programs and in Paper II, the study was performed on 36 programs. The programs were collected from SourceForge [29], a common source for open source software at the time, selected for its easy to use search function. In both cases, the 100 most popular Python programs were collected and compared to a number of criteria to include them or exclude them from the study. The selection was made at different times which resulted in different sets of programs for the two studies. The first requirement was that the programs were entirely developed in Python and contained no parts written in other languages. The reason for this was that the instrumentation only worked for Pyhton code and no data could be collected from program parts written in other languages. The programs also had to be actively developed at the time. Another criterium used for selection was that the programs were "useful", that is used by others than the developers with > 1000 downloads. This criterium was used to sort out *e.g.*, uploaded solutions to typical educational programming assignments. The programs also had to be tagged with "production/stable" to secure that the developers considered the program to be mature enough for others to use. The instrumented Python interpreter used to collect the data was Python version 2.6 and run under Debian GNU/Linux, which meant that the progams used in the study also had to run in this setting. The programs also needed to be possible to use without special hardware (e.g., microscopes or cameras) subscriptions or specialist knowledge in the application domain. To be able to document the running of all programs in the same way in terms of use-cases, the programs included in Paper I were all interactive programs with a graphical user interface. This requirement was omitted for the program selection for Paper II.

4.1.4 Threats to Validity

The number of selected programs is low in both Paper I (19) and Paper II (36) and the collection was not made randomly, which introduces the risk that they are not representative for all Python programs and that general conclusions cannot be made. The programs were however not connected, they do not solve the same problems, are not from the same application area and were not produced by the same group of people. This leads us to the conclusion that the results from our studies are valid for many Python programs.

As discussed in the "Threats to Validity" section of Paper II, the initial analysis was made on individual call-sites, which is a fine-grained analysis which is likely to over-estimate typability. To remedy this, we collected call-sites in clusters of connected call-sites where a cluster was defined as a set of call-sites from the same source file, connected by the receivers they have seen. The analysis made on clusters, however, comes with its own limitations. First of all it may be too coarse-grained and under-estimate the typability in cases when value-based overloading has been used. The conclusions drawn in the paper suggests that the results of the analysis made on individual call-sites may be used as an upper bound and the results from the analysis made on clusters may be used as a lower bound on the typability. If the clustering had instead been made using a data-flow analysis separate clusters would have been collected for different runs of the method `vbo()` in Listing 4.1, either containing the call-sites on lines 14 and 16 or lines 14 and 18. Since the call site on line 14 appears in both, all three call-sites would still have to be typed together to make sure that the type of the call-site on line 14 works in both cases. The groups of call-sites analysed together would be the same as in the paper's cluster analysis, limited by the expressiveness of the data-flow analysis.

In TypeScript [78], a slightly modified version of the program in Listing 4.1 could have been typed using a *union type*, which allows the value to be of one of a set of types. The conditional would in that case include an `instanceof` to make sure which one of the listed types is currently the type to make type system accept operations specific for that type.

In previous work on type inference in Self, templates were generated for polymorphic versions of methods [1]. To avoid generating an exponential number of templates, polymorphism was measured in a number of Self programs and the results showed that 0-4 % of the methods were megamorphic if the the limit between polymorphic and megamorphic calls were set to 3. The limit in their analysis was set to a number between 3 and 5. In Paper II we followed this example and set the limit between polymorphic and megamorphic to 5, that is megamorphic call sites had been logged with more than 5 different receiver types. In our results, the share of call-sites that were only logged

once were 50 % on average for all programs and the share of monomorphic call-sites were 46 %. The share of polymorphic call sites with two different receiver types were 2.6 % and the share of polymorphic call sites with a higher number of receiver types (2-5) and the megamorphic call sites were all below 1 %. These figures do not suggest that the results of the analysis would have been different if the limit between polymorphic and megamorphic calls would have been set higher or slightly lower.

4.1.5 What Has Happened In Dynamic Languages?

Since Paper I and Paper II were published in 2014 and 2016, the work on combining dynamically typed languages with static type systems has evolved. The research community has continued the work on understanding the difficulties arising from how programmers use dynamically typed languages in practice, *e.g.*, [101–109].

As in Paper I and II, Kaleba et al. [101] use an instrumented version of a language runtime to study call-site behaviour, but their work is made on Ruby. Their work focuses on call-site-related optimisations in the runtime environment rather than typability.

Chen et al. [102; 103] identifies dynamic programming patterns that are prone to cause type-related errors in Python programs (`TypeError` and `AttributeError`). Some of the "type smells" identified are related to Python's dynamic features studied in Paper I, *e.g.*, Dynamic Attribute Deletion and Dynamic Attribute Access, confirming that these features are used in real-world Python programs.

Sun et al. [104] study object dynamism in 50 mature Python programs. They separate object dynamism into object evolution (changes made to an object's type after creation) which was studied in Paper I, and constructor polymorphism (one constructor producing objects of more than one type). Their study confirms our results from Paper I, that object evolution is common and includes both attribute addition, modification and deletion. They also discuss the differences between using class-based types and object-based types for typing dynamic objects, which is a similar discussion but using other alternatives than in Paper II where the comparison was made for nominal and structural types.

Yang et al. [105] build on the results from Holkner and Harland [13] and Paper I aiming to build better static analysis tools for Python. They vastly extend the set of programs studied and include also some more Python features. Since their focus is on complex features rather than dynamic ones, they include also *e.g.*, *list comprehensions* and *decorators*. Their analysis is static, made on the syntax tree and confirms our result in that dynamic features are common

in real-world programs. Their static analysis is also combined with a small random sample of manual source code inspections aiming to understand the use of complex features and also identify patterns that can or cannot be handled by static analysis.

The use of new hybrid languages has also spread, *e.g.*, TypeScript (first released in 2012) and Hack (first released in 2014). TypeScript is listed as number 41 in the TIOBE list of the most popular programming languages in September 2024, and Hack is one of the languages listed among number 50-100 [110].

Typescript is a gradually typed extension of JavaScript and the TypeScript compiler outputs JavaScript code. All JavaScript programs are also valid TypeScript programs. The type system uses structural types and in line with the results presented in Paper II, the type system of TypeScript allows code that cannot be determined safe at compile-time [111].

Preceded by long and intense dicussions in the Python community, the possibility to annotate functions with unchecked argument and return type annotations [112] was added in Python[1] version 3.0 (released in 2006). The annotatons were often used by external libraries to perform static analyses on Python code, but the annotations were not explicitly stated to be used that way. Type hints [84] were added in Python version 3.5 (released in 2015) to provide a standard way of expressing type information. The type hints have since evolved but they are still optional.

The type hints may be used by third-party tools to check types statically, andseveral different tools are available to do this, *e.g.*, [79; 85; 86]. Another option is to use Python's built-in reflection mechanisms to access the type annotations to improve the support for dynamic type checking as *e.g.*, in [113].

4.2 Improving the Support for Programming with Arrays

To answer the research questions from Theme II, I have performed three studies presented in Paper III, Paper IV and Paper V.

Types have emerged as a promising way of making concurrent and parallel programming easier for programmers. In the *Kappa* type system [14], annotations embedded in types control what techniques will be used to guarantee data-race freedom or atomicity in certain parts of a program. This allows programmers to capture their intentions (*e.g.*, how/if a value is shared, or accessed) in code, and have those intentions verified statically by a compiler. Kappa was developed for object-oriented languages to operate at the level of

[1]Python should be understood as CPython, the reference implementation of the Python programming language.

objects–not arrays. As a consequence, the same concurrency control mechanism (*e.g.*, locks, linearity, etc.) is applied to both the array and all objects reachable from its elements. This either gives rise to very coarse-grained concurrency (you either have access to the whole array and all of its elements, or nothing) or very fine-grained concurrency (access to every element is managed separately), but nothing in between. Kappa excludes data-races meaning write access to an object can only be granted to one thread at a time. Sadly, this excludes parallel operations inside arrays that are not immutable. Divide and conquer-style parallel algorithms commonly split arrays into subparts that can be accessed concurrently. Such treatment of arrays is not supported by Kappa which prohibits parallelisation of array-based algorithms. (Large parts of the text in this section was taken from earlier work presented in Paper VI listed under "Related articles" in Section 1.2.)

Included in this dissertation are articles that present and evaluate *array capabilities*, an extension to reference capabilities which grant access to (parts of) an array, while absence of data races is still guaranteed statically. Array capabilities allow programmers to implement parallel array algorithms at a higher level of abstraction by using the provided split and merge operations rather than individual indices. We illustrate how split and merge operations using array capabilities can be used in practice and we show that arrays in real world programs are to a great extent accessed using uncomplicated patterns *e.g.*, traversals that are easily modeled by using array capabilities.

4.2.1 Paper III: Array Capabilities

We present array capabilities, an extension to reference capabilities that allow arrays to be subdivided into array slices, but without copying the content. Similar to slices in Go and Rust, an array capability provides a view into an underlying array. Unlike

Like Rusts slices (see section 3.2.3), an array capability allows more than one array capability to access the same underlying array and modifications may be allowed under certain circumstances. In Rust this requires that the slices are created by splitting the original array at a specified index. Array capabilities may be split using a more flexible set of splitting operation, while still maintaining the guarantee that changes made through one of them will never have any impact on other potentially existing array capabilities.

In the same way as reference capabilities an an array capability is an unforgeable token that governs access to a resource, which in this case is an array. Like reference capabilities, array capabilities come with guarantees on what concurrent holders of overlapping array capabilities cannot do in terms of access to elements or contents stored in elements. If more than one array ca-

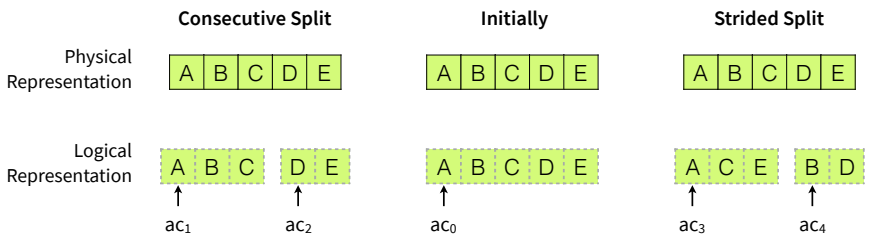


Figure 4.10: Splitting array capabilities. The middle figure shows a possible initial state, *e.g.*, after the creation of the array. The physical representation, that is the underlying array, and the logical representation, that is the array capability `ac0` in this state are identical. The figures to the left and right show possible results of using split operations on `ac0`. The left one shows the result of a consecutive 2-split—dividing the array capability into two consecutive parts (`ac1` and `ac2`) of the original. The right figure shows the result of a strided 2-split, where the two subarrays (`ac3` and `ac4`) were created from the odd and even elements of the array respectively. The underlying array remains the same and unchanged both to the left and to the right.

pability give views into the same underlying array, the type systems guarantee that no part of the underlying array is part of more than one array capability.

Following the fundamental design of *Kappa*, where the full interface of a capability always can be used if it can be created, the design of array capabilities always grant access to all all elements of an array or sub-array. Subarrays accessed through array capabilities are thus indexed from zero and are indistinguishable from arrays. Any code operating on arrays will work also for array capabilities, although accessing adjacent indexes of a sub-array may be translated to non-adjacent indexes of the underlying array.

Sub-arrays are created by using the split operation available in the array capability (which could, in turn, already be a sub-array). The split operation can be performed consecutively or in a strided fashion. An example showing the split operation at work can be found in Figure 4.10.

The middle figure in Figure 4.10 shows an initial state where the array capability ac_0 provides a view of an underlying array which is shown above the array capability. In this case, the order of elements in the array capability and the underlying array are identical.

To the left, are two array capabilities created by a consecutive split, ac_1 and ac_2, which both provides views of the same array capability. ac_1 shows element 0-2 in the underlying array and ac_2 shows element 3-4 in the underlying array (but ac_2 is indexed from 0 to 1).

The example to the right instead shows two array capabilities created by a strided split, ac_3 and ac_4 that show views to the same array capability. In this

case, ac_3 contains all even indexes from the underlying array and ac_4 contains all odd indexes. Just as in the array capability $ac_0 2$, ac_3 and ac_4 are indexed from 0-2 and 0-1, respectively.

The splitting into subarrays is all logical, meaning that no actual creation of array objects takes place. The only extra run time work needed is translating the indexes using meta-data about the sub-array that can easily be placed on the stack.

All array capabilities guarantee the absence of data races in a well-typed program.

4.2.2 Paper IV: Using Array Capabilities

To validate the work presented in Paper III and to gain a better understanding of the design requirements for array capabilities, I developed a prototype implementation. The prototype was implemented in Python to keep focus on the design of the array capability API. To evaluate the prototype implementation, I then used it to implement several existing array and matrix algorithms. The code from array and matrix algorithm implementations were compared to code from corresponding implementations made in Python. The comparisons made were both quantitative (*e.g.*, by counting the number of direct index manipulations needed) and qualitative (*e.g.*, manually reading the code to identify improvement opportunities for the API).

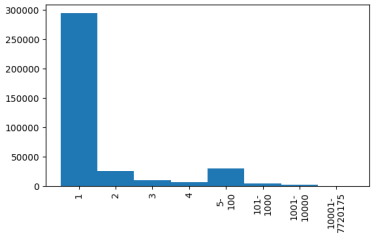
In addition to identifying the need for additional operations, it was examined what features are commonly exercised, what are the recurring patterns, and how reliance on direct element addressing using indexes can be reduced.

As a result of this study:

- We identified some new convenient splitting operations ("split at" and "split by").
- We confirmed that the splitting operations already defined for array capabilities are useful.
- We realised that the merging operations are less important when borrowing is applied to the splitting operations.
- The dependence on direct index-based accesses decreased in general in our implementations.

4.2.3 Paper V: Array Access Patterns

The array capability presented in Paper III, and evaluated through a prototype in paper IV, supports reordering the elements of the underlying array to better match the requirements in actual use *e.g.*, for better cache performance.



#arrays	#access patterns
1	295,129
2	25,524
3	10,549
4	6,835
5-100	30,481
101-1000	4,707
1001-10000	1,790
10001-7720175	531

Figure 4.11: Each array has been accessed in a certain pattern. The number of arrays sharing the same access patterns can be seen in the histogram and the table. 295,129 access patterns represent one single array, 25,524 access patterns represent two arrays and so on. 30,481access patterns represented between 5 and 100 arrays, 4,707 access patterns represented between 101 and 1,000 arrays and so on.

Through the work presented in Paper V we got further input in our continued work on array capabilities by looking at how arrays are used in real-world programs.

We collected array access patterns from real-life programs running on the JVM by instrumenting the programs to log all array accesses. The data collected included array sizes, element types, from where arrays are accessed (class and thread), the indexes accessed and whether the access was a read or a write operation. The 25 (7 Java and 18 Scala) programs in the study were collected from the Renaissance benchmark suite, all running on the JVM.

After the data was collected, all access patterns were identified. Access patterns are only concerned with the access order, the thread identity, and the mode of the operation (read or write) and the length of the array or the type of the elements are not considered. The access pattern may be unique or be shared by many arrays and arrays with different lengths and storing different types of elements could be accessed using the same access pattern. The number of arrays that share the same access pattern can be found in Figure 4.11.

The access patterns were examined using an iterative process to see if they (or parts of them) matched an identified *shape*. The *shapes* can be seen in Figure 4.12, where the x axis represents the time, the y axis represents the indices accessed. The full colour represents write accesses and the more transparent colour represents read accesses.

The result of the search for identified shapes can be seen in Figure 4.13. There is one pie chart per program and in each pie chart the share of access patterns that consisted entirely of identified shapes is blue, the share of access patterns that consist partially of identified shapes is orange and the share of access patterns where no identified shapes could be identified is green. On average, for all benchmarks, 81.4 % of the access patterns consisted entirely of

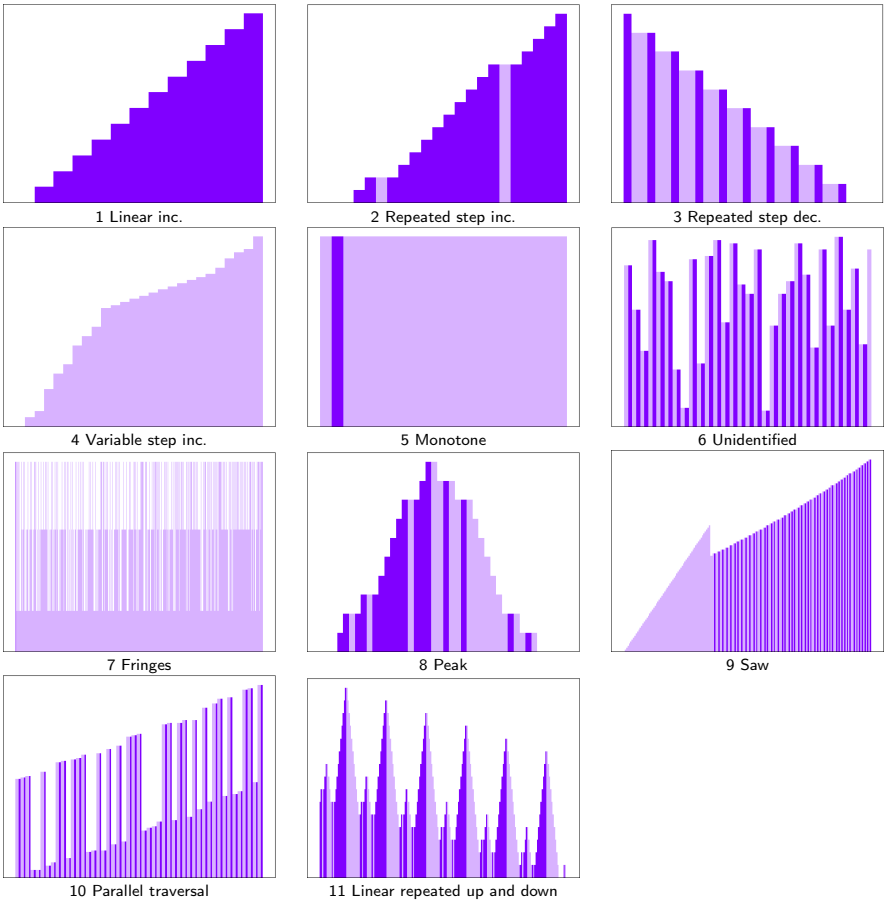


Figure 4.12: Examples of array some access pattern shapes identified by manual inspection, and in example 6 an access pattern shape that is unidentified.

identified shapes. The benchmark with the lowest share of entirely identified shapes was `scala-stm-bench7` (45.6 %) The benchmark `scala-doku` had the largest share of access patterns that contained no identified shapes (25.4 %). On average, for all benchmarks, 7.2 % of the access patterns contained no identified shapes.

The distribution of individual accesses distributed over the access pattern shapes can be seen in Figure 4.14. The dominating shape is the "repeated step increasing", which incorporates 30.8 % of all accesses. The second most common identified shape is the "linear increasing" which covers 23.0 % of all accesses.

This shows that access patterns consist of traversals to a great extent; the access pattern repeated step increasing together with linear increasing incor-

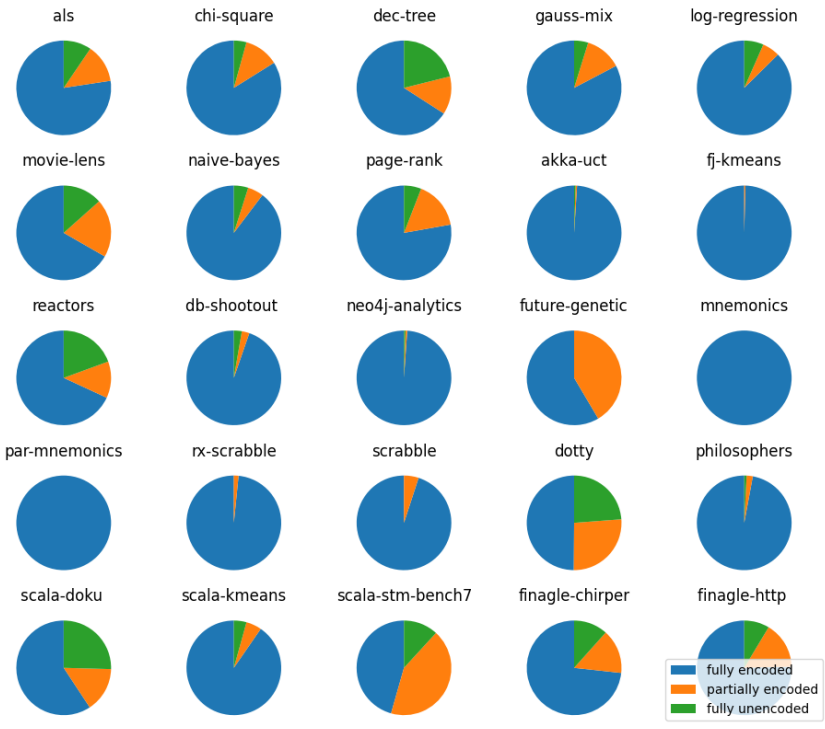


Figure 4.13: The result of searching for identified shapes in the access patterns.

porates 53.8 % of all accesses. This means that the indices for more than half of the array accesses could be controlled by a for loop. 15.1 % of all accesses are however still made in patterns that are unidentified.

4.2.4 Threats to Validity

In Paper V, the shapes of access patterns were identified using an iterative approach where access patterns were plotted and inspected manually and access patterns matching common shapes were selected and sorted out to enable identification of less common shapes in the next iteration. This method is not exact and an approach using machine learning could probably have been used to improve the identification process, especially in identifying more complex shapes. The method used however succeeded in identifying access pattern shapes that could be used to cover 81.4 % of all the access patterns.

Access pattern shape	% of all accesses
Repeated step increasing	30.8 %
Linear increasing	23.0 %
Saw	8.1 %
Constant	7.5 %
Linear decreasing	7.5 %
Peak	4.1 %
Linear repeated up and down	1.7 %
Fringes	0.7 %
Variable step decreasing	0.3 %
Parallel traversals	0.3 %
Repeated step decreasing	0.1 %
Variable step decreasing	0.0 %
Unidentified	15.1 %

Figure 4.14: Distribution of individual array accesses over patterns.

5. Conclusions and Future Work

This chapter presents the collective conclusions drawn from the papers included in this thesis and discusses some alternative methods. The chapter is finished off by a discussion of possible paths for future work.

5.1 Conclusions

The general research question for the work presented in this dissertation is "To what extent can we use static types to capture dynamic behaviour?" and to examine this I have broken this down into two main themes to understand and to improve typability:

Theme I: Understanding how types are used in programs written in dynamically typed languages

The research questions on the first theme were:

Do programmers actually use dynamic features available in dynamic programming languages that make changes to types possible at run time?

Presented in Paper I, I have show that the use of language constructs that potentially make changes to existing types is commonly found in Python programs (Python 2.6). Their presence is sometimes sparse, but they are found in both library code (69 % of all dynamic features used) and program-specific code (31 % of all dynamic features used). They are also distributed over program start-up (55 %) and over the entire run time (45 %).

This leads to the conclusion that it is common that Python programs make use of dynamic features to exhibit behaviours which are inherently hard to type.

Do programmers write programs using polymorphic variables?

The results of Paper II show that the actual call-site receiver types present at run time are to a great extent monomorphic (96 % on average), which means that large parts of programs could be typed using simple type systems. The remaining (4 %) however contains call-sites where up to 356 different receiver types were logged for a single call-site and they can to a great extent not be

typed using nominal or structural systems, for example due to use of value-based overloading.

We conclude that large parts of Python programs consist of code that could be typed *e.g.*, with a simple nominal type system, but that it is common that programs contain small parts that are not possible to type using a simple type system.

Conclusions from Theme I The overall conclusion from the two studies made on this theme is that program written in dynamic programming languages do exhibit dynamic behaviours, both in terms of the use of dynamic features and polymorphism, but that large parts of each program contain code that could be typed *e.g.*, with a simple nominal type system.

We conclude that a static type system used on Python code must allow dynamic behaviour in arbitrary parts of the program and throughout the program's run time. This provides support for the case of gradual typing where large parts could be statically typed but still letting parts of programs that are highly dynamic be left without static types.

Theme II: Improving support for programming with arrays

The research questions on the second theme were:

How can concurrent programming patterns involving arrays be checked statically?

In Paper III, I have presented an extension to reference capabilities, the array capability, to support race-condition free concurrent and parallel operations on arrays.

Are array capabilities useful in practice?

In Paper IV, I have presented the evaluation of a prototype implementation of the array capabilities. The evaluation resulted in some potential improvment suggestions to make the programming API more expressive and it showed that the reliance on explicit indexes was decreased in the implemented programs.

Are arrays used in regular patterns that allow splitting according to the operations defined for array capabilities?

In Paper V I have shown that arrays are often used in ways that would present no problems to type using array capabilities, but further studies of arrays used in parallel are needed to fully understand the needs and limitations present in real-world programs.

Conclusions from Theme II In Paper III, we presented array capabilities, a new type of reference capabilities to support *e.g.*, logical splitting and merging

of arrays. The array capabilities are presented in a parallel calculus and we prove that they guarantee freedom from data races.

From Paper IV we conclude that the array capabilities presented in Paper III are useful when implementing algorithms operating on arrays. The decreased dependence on direct index based accesses supports the programmer in focusing on algorithms at a higher level of abstraction.

From Paper V we conclude that array capabilities may be used to model large parts of array algorithms used in real world programs.

From the work done on this theme, we conclude that static types can be extended to support programmers in writing parallel programs operating on arrays.

Conclusions From the work presented in this thesis we draw the conclusions that static types can be used to capture dynamic behaviour to a great extent, but not entirely. Types can be used to provide programmers with more support when writing complex code, *e.g.*, in array algorithms. Array capabilities can be used to type arrays most array accesses but not all.

Using static types allows us to statically detect and reject incorrect programs, but omitting static types makes it possible to write correct programs that could not be proven right. Without the static types the programmer will however lack the type system's support in finding errors.

In the end, this means that we have achieved a better understanding for the diversity and dynamicity of types in dynamic languages and that we have taken a first step towards understanding the typability of array access patterns. Programming languages that need to be backward compatible must allow programmers to opt out from the static type checking.

5.2 Alternatives to Typability

In the work presented in this dissertation, types serve as a method for examining and analysing programs. Types are used to statically express properties of the running program. A program that cannot be successfully type-checked, a program that contains errors from the type system's point of view, would potentially result in incorrect behaviour. The analysis is conservative and may in itself be incorrect. There are other alternatives for program verification than using static type systems, such as *e.g.*, dataflow analysis [114], abstract interpretation [114], static analysis tools (*e.g.*, Infer [115]), dynamic program analysis and program verification techniques, such as separation logic [116].

Dataflow analysis uses a control-flow graph to track the possible paths a value may propagate through the program. It keeps track of all variables de-

fined in the program and all their possible values. Dataflow analysis is commonly used by compilers to detect possible optimisations. As opposed to analyses using type systems, where many possible runs of the programs may be ruled out due to typing rules, the dataflow analysis has to account for them all.

Dynamic program analysis includes techniques like *e.g.*, functional testing, profiling and memory error detecting tools like Valgrind [117]. These techniques are usually applied to programs that have already been analysed by a type system, but by using tools that specifically search for errors that cannot be detected by a static type system.

Static analysis tools, as *e.g.*, Infer [115], are separate tools and not part of the programming language. They can be used to detect a broader range of errors but may not be used for analysis as early in the development process as a type system. They can be applied on program parts (*e.g.*, functions and modules) but not on individual lines of code.

Formal systems like separation logic are useful *e.g.*, for proving correctness for parallel array algorithms but would require huge efforts to provide proofs for large-scale systems. They also require highly specialised skills.

An advantage of using a type system as a light-weight static analysis is that the tool is built into the language and consequently is available to the programmer at all times. Since type systems are integrated into the programming language, programmers learn to use them when learning to use the language. To perform the analysis no special training is required. Using type systems also provides scalable analyses for full programs.

5.3 Future Work

We see many interesting venues for future work.

In the study of dynamic features, no thorough inspection of the code was made to understand why these features were used or if the use could be judged as justified in the context. We would like to follow this up by a qualitative investigation of the usage patterns for dynamic features in Python. The goals of such an investigation would be both to understand situations where the use is meaningful and also to investigate to what extent they could be removed by refactorings, to yield code which would be easier to type statically.

We would like to further our empirical studies of arrays by developing the analysis method to study more complicated access pattern shapes. An approach that we would like to use to identify shapes is by using machine learning for pattern recognition. Patterns of special interest for future studies are, *e.g.*, patterns that consist of interleaved traversals, and patterns with accesses made from two or more parallel threads. We expect that such interesting patterns can be found in scientific computing programs.

An implementation of array capabilities would allow further exploration of the array splitting and merging design space. One focus of the implementation will be to design a better index translation mechanism focused on performance.

Extending the array access analysis to also cover the life-time of arrays and the identity of objects accessing the arrays, besides the class identity.

Sammanfattning

Vid programutveckling använder programmerare många olika verktyg för att försäkra sig om att koden motsvarar den förväntade funktionaliteten i programmet: testverktyg, felsökningsprogram, och så vidare. Det är också vanligt att programmeringsspråk tillhandahåller ett statiskt typsystem som används för att kontrollera att alla operationer som utförs i programmet är meningsfulla. Målet med arbetet som presenteras i denna avhandling är att studera typbarhet från olika perspektiv. Typbarhet beskriver möjligheten att hitta en precis typ för ett givet uttryck, en typ som är tillräckligt specifik för att kunna hjälpa programmeraren och kompilatorn i arbetet med att analysera koden för att hitta felaktigheter och optimeringsmöjligheter, men fortfarande såpass flexibel att den är användbar i praktiken. För att uppnå detta mål har jag undersökt hur typer används i program skrivna i dynamiskt typade språk och även hur typer kan vidareutvecklas för att tillhandahålla utökat stöd till programmerare i paralellexekverande program. Slutmålet för detta är att bidra till skapande av programmeringsspråk som använder statiska typer för att stötta programmeraren när stödet behövs, samtidigt som programmeraren tillåts frihet att använda sin skicklighet för att skriva program som inte kan ges statiska typer. Det vill säga skriva fungerande program som ett typsystem inte kan garantera frånvaro av felaktigheter.

Den övergripande forskningsfrågan i projektet har varit ”i vilken utsträckning kan statiska typer användas för att fånga dynamiskt beteende?” och arbetet har utförts inom två olika teman.

Tema I: Förstå hur typer används i program skrivna i dynamiska språk
På detta första tema har jag genomfört två empiriska studier för att förstå typbarhet. Jag har undersökt i vilken utsträckning program skrivna i dynamiskt typade språk faktiskt utnyttjar den dynamik som språket tillåter. Dynamiska språk tillåter vanligtvis användning av mycket dynamisk funktionalitet som omdefinition av klasser (och till och med enskilda objekt) under körning och obegränsad polymorfism, det vill säga att tillåta samma kod att fungera på olika typer utan några begränsningar. För att undersöka detta har jag utforskat möjligheterna att i efterhand applicera ett statiskt typsystem på program skrivna i ett dynamiskt språk.

Tema II: Förbättring av stöd för programmering med arrayer På det andra temat har jag använt både teoretiska och empiriska metoder för att utveckla och utvärdera mer precisa typer för arrayer som stödjer programmeraren genom att förhindra kapplöpningstillstånd i parallellexekverande program. I det teoretiska arbetet har jag presenterat “array capabilities”, en array-typ som stödjer alias-kontroll och garanterar trådsäkra samtidiga operationer på arrayer. Det empiriska arbetet genomfördes för att möjliggöra vidareutveckling av det teoretiska arbetet. Dels genomförde jag en undersökning av hur arrayer faktiskt används i verkliga program, dels implementerade jag en prototyp av “array capabilities” och gjorde en praktisk utvärdering.

De slutsatser jag har dragit på det första temat, Förstå hur typer används i program skrivna i dynamiskt typade språk", är sammanfattningsvis att program skrivna i dynamiskt typade språk inte i efterhand kan passas in i ett statiskt typsystem. Programmerare skriver dock kod som i stor utsträckning är typbar, även när de inte tvingas till det av ett statiskt typsystem. Detta stärker stödet för “gradual typing”, där delar av ett program kan förses med statiska typer medan andra, mer dynamiska, delar av samma program lämnas utan statiska typer.

De slutsatser som dragits på det andra temat, Förbättring av stöd för programmering med arrayer", är sammanfattningsvis att statiska typer kan utökas för att stödja programmerare i att skriva parallella program som arbetar med arrayer.

Som helhet är slutsatsen att vi kan använda statiska typer för att fånga dynamiskt beteende i stor utsträckning, men inte helt. Typer kan användas för att ge programmerare mer stöd när de skriver komplex kod, t.ex. i arrayalgoritmer, men språket bör också tillåta programmeraren att skriva program som innehåller beteende som inte kan fångas av statiska typer.

References

[1] OLE AGESEN. **The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism**. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, page 2–26, Berlin, Heidelberg, 1995. Springer-Verlag. 19, 61, 70

[2] JONG-HOON (DAVID) AN, AVIK CHAUDHURI, JEFFREY S. FOSTER, AND MICHAEL HICKS. **Dynamic inference of static types for ruby**. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 459–472, New York, NY, USA, 2011. Association for Computing Machinery.

[3] DAVIDE ANCONA, MASSIMO ANCONA, ANTONIO CUNI, AND NICHOLAS D. MATSAKIS. **RPython: a step towards reconciling dynamically and statically typed OO languages**. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, page 53–64, New York, NY, USA, 2007. Association for Computing Machinery.

[4] JOHN AYCOCK. **Aggressive Type Inference**. In *Proceedings of the 8th International Python Conference*, pages 11–20, 2000.

[5] MICHAEL FURR, JONG-HOON (DAVID) AN, JEFFREY S. FOSTER, AND MICHAEL HICKS. **Static type inference for Ruby**. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, page 1859–1866, New York, NY, USA, 2009. Association for Computing Machinery.

[6] BRIAN HACKETT AND SHU-YU GUO. **Fast and precise hybrid type inference for JavaScript**. *SIGPLAN Not.*, **47**(6):239–250, jun 2012.

[7] MICHAEL SALIB. *Starkiller: A Static Type Inferencer and Compiler for Python*. Master's thesis, Massachusetts Institute of Technology, 2004.

[8] PETER THIEMANN. **Towards a Type System for Analyzing Javascript Programs**. In *Proceedings of the 14th European Conference on Programming Languages and Systems*, ESOP'05, page 408–422, Berlin, Heidelberg, 2005. Springer-Verlag.

[9] SAM TOBIN-HOCHSTADT AND MATTHIAS FELLEISEN. **Interlanguage migration: from scripts to programs**. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 964–974, New York, NY, USA, 2006. Association for Computing Machinery. 19, 61

[10] SYLVAIN LEBRESNE, GREGOR RICHARDS, JOHAN ÖSTLUND, TOBIAS WRIGSTAD, AND JAN VITEK. **Understanding the Dynamics of JavaScript**. In *Proceedings for the 1st Workshop on Script to Program Evolution*, STOP '09, page 30–33, New York, NY, USA, 2009. Association for Computing Machinery. 19

[11] GREGOR RICHARDS, SYLVAIN LEBRESNE, BRIAN BURG, AND JAN VITEK. **An analysis of the dynamic behavior of JavaScript programs**. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 1–12, New York, NY, USA, 2010. Association for Computing Machinery. 19

[12] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. **How Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk**. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, page 23–32, New York, NY, USA, 2011. Association for Computing Machinery. 19

[13] Alex Holkner and James Harland. **Evaluating the Dynamic Behaviour of Python Applications**. In *In Proceedings of Thirty-Second Australasian Computer Science Conference (ACSC 2009)*, pages 17–25, Wellington, New Zealand, 2009. 19, 21, 71

[14] Elias Castegren and Tobias Wrigstad. **Reference Capabilities for Concurrency Control**. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *ECOOP 2016*, **56** of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:26, Dagstuhl, Germany, 2016. 19, 43, 72

[15] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. **Uniqueness and reference immutability for safe parallelism**. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 21–40, New York, NY, USA, 2012. Association for Computing Machinery. 30

[16] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. **Deny capabilities for safe, fast actors**. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2015, page 1–12, New York, NY, USA, 2015. ACM. 19, 42

[17] Yuan Lin and David A. Padua. **Demand-Driven Interprocedural Array Property Analysis**. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '99, page 303–317, Berlin, Heidelberg, 1999. Springer-Verlag. 20, 29

[18] Thi Viet Nga Nguyen and François Irigoin. **Efficient and effective array bound checking**. *ACM Trans. Program. Lang. Syst.*, **27**(3):527–570, may 2005.

[19] William Pugh. **A practical algorithm for exact array dependence analysis**. *Commun. ACM*, **35**(8):102–114, aug 1992.

[20] Yunheung Paek, Jay Hoeflinger, and David Padua. **Simplification of Array Access Patterns for Compiler Optimizations**. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, page 60–71, New York, NY, USA, 1998. Association for Computing Machinery.

[21] W. Blume and R. Eigenmann. **Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs**. *IEEE Trans. Parallel Distrib. Syst.*, **3**(6):643–656, nov 1992. 20, 29

[22] Beatrice Åkerblom, Jonathan Stendahl, Mattias Tumlin, and Tobias Wrigstad. **Tracing dynamic features in Python Programs**. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 292–295, New York, NY, USA, 2014. ACM. 20

[23] Beatrice Åkerblom and Tobias Wrigstad. **Measuring Polymorphism in Python Programs**. *ACM SIGPLAN Notices*, **51**(2), February 2015. 21

[24] Beatrice Åkerblom, Elias Castegren, and Tobias Wrigstad. **Reference Capabilities for Safe Parallel Array Programming**. *The Art, Science, and Engineering of Programming*, **4**(1), May 2019. 22

[25] John Boyland, James Noble, and William Retert. **Capabilities for Sharing**. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming*, pages 2–27, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. 22, 42

[26] Beatrice Åkerblom, Elias Castegren, and Tobias Wrigstad. **Progress Report: Exploring API Design for Capabilities for Programming with Arrays**. In *Proceedings of the 14th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '19, New York, NY, USA, 2019. ACM. 23

[27] Beatrice Åkerblom and Elias Castegren. **Arrays in Practice–An Empirical Study of Array Access Patterns on the JVM**. *The Art, Science, and Engineering of Programming*, **8**(3), 2024. 24

[28] Anjana Gosain and Ganga Sharma. **A Survey of Dynamic Program Analysis Techniques and Tools**. In Suresh Chandra Satapathy, Bhabendra Narayan Biswal, Siba K. Udgata, and J.K. Mandal, editors, *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*, pages 113–122, Cham, 2015. Springer International Publishing. 29

[29] Slashdot Media Source Forge. **Source Forge Open Source Software**. https://sourceforge.net/, 2014. Accessed: 2014. 29, 69

[30] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. **Renaissance: Benchmarking Suite for Parallel Applications on the JVM**. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 31–47, New York, NY, USA, 2019. ACM. 29

[31] Youn-kyung Lim, Erik Stolterman, and Josh Tenenberg. **The anatomy of prototypes**. *ACM Transactions on Computer-Human Interaction*, **15**:1–27, 07 2008. 30

[32] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2016. 34, 35, 39, 52

[33] John Backus. *The History of Fortran I, II, and III*. ACM, New York, NY, USA, 1978. 34, 44

[34] John McCarthy. **History of LISP**. In *History of Programming Languages*, page 173–185, New York, NY, USA, 1978. ACM. 34, 44

[35] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger, and P. Naur. **Revised Report on the Algorithmic Language ALGOL 60**. *Communications of the ACM*, **6**(1):1–17, jan 1963. 34

[36] George Radin. **The early history and characteristics of PL/I**. *SIGPLAN Not.*, **13**(8):227–241, aug 1978. 34

[37] D. L. Parnas. **A technique for software module specification with examples**. *Commun. ACM*, **15**(5):330–336, may 1972. 34

[38] Barbara Liskov and Stephen Zilles. **Programming With Abstract Data Types**. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, page 50–59, New York, NY, USA, 1974. Association for Computing Machinery. 34, 53

[39] Bertrand Meyer. *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., USA, 1997. 34

[40] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990. 34

[41] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. 34

[42] JOSEPH ALBAHARI AND BEN ALBAHARI. *C# 5.0 in a Nutshell: The Definitive Reference*. O'Reilly Media, 5th edition, 2012. 34, 48

[43] ORACLE. **Java Platform, Standard Edition–Java Language Updates**. https://docs.oracle.com/en/java/javase/21/language/index.html, 9 2023. Release 21. 34

[44] JENS PALSBERG AND MICHAEL I. SCHWARTZBACH. *Object-oriented type systems*. Wiley professional computing. Wiley, 1994. 35

[45] BENJAMIN C. PIERCE. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. 35, 44

[46] B. MEYER. **Applying 'Design by Contract'**. *Computer*, **25**(10):40–51, 1992. 35, 40

[47] AMER DIWAN, KATHRYN S. MCKINLEY, AND J. ELIOT B. MOSS. **Type-Based Alias Analysis**. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, page 106–117, New York, NY, USA, 1998. ACM. 35

[48] DAVID G. CLARKE, JOHN M. POTTER, AND JAMES NOBLE. **Ownership types for flexible alias protection**. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, page 48–64, New York, NY, USA, 1998. ACM. 35, 40, 41, 42

[49] CHANDRASEKHAR BOYAPATI, ROBERT LEE, AND MARTIN RINARD. **Ownership Types for Safe Programming: Preventing Data Races and Deadlocks**. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, page 211–230, New York, NY, USA, 2002. Association for Computing Machinery. 35

[50] BARBARA H. LISKOV AND JEANNETTE M. WING. **A Behavioral Notion of Subtyping**. *ACM Trans. Program. Lang. Syst.*, **16**(6):1811–1841, nov 1994. 35

[51] ROBERT W. SEBESTA. *Concepts of Programming Languages*. Pearson, 10th edition, 2012. 36

[52] WILLIAM LANDI AND BARBARA G. RYDER. **Pointer-induced aliasing: a problem classification**. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, page 93–103, New York, NY, USA, 1991. Association for Computing Machinery. 38

[53] JOHN HOGG, DOUG LEA, ALAN WILLS, DENNIS DECHAMPEAUX, AND RICHARD HOLT. **The Geneva Convention on the Treatment of Object Aliasing**. *ACM SIGPLAN OOPS Messenger*, **3**(2):11–16, apr 1992. 39, 40

[54] JOHN HOGG. **Islands: aliasing protection in object-oriented languages**. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '91, page 271–285, New York, NY, USA, 1991. ACM. 39, 41

[55] MICHAEL HIND. **Pointer analysis: haven't we solved this problem yet?** In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, page 54–61, New York, NY, USA, 2001. Association for Computing Machinery. 39

[56] AMROZ SIDDIQUI AND KALIAPPAN YADAV. **A Novel Approach to Pointer Analysis**. In *2022 OPJU International Technology Conference on Emerging Technologies for Sustainable Development (OTCON)*, pages 1–5, 2023. 39

[57] STEPHAN BRANDAUER AND TOBIAS WRIGSTAD. **Spencer: Interactive Heap Analysis for the Masses**. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 113–123, 2017. 40

[58] JANINA VOIGT AND ALAN MYCROFT. **Aliasing Contracts: A Dynamic Approach to Alias Protection**. Technical Report ISSN 1476-2986, University of Cambridge, 2013. UCAM-CL-TR-836. 40

[59] AARON GREENHOUSE AND JOHN BOYLAND. **An Object-Oriented Effects System**. In RACHID GUERRAOUI, editor, *ECOOP' 99 — Object-Oriented Programming*, pages 205–229, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. 40

[60] JOHN BOYLAND. **Alias burying: unique variables without destructive reads**. *Softw. Pract. Exper.*, **31**(6):533–553, may 2001. 40, 41

[61] JOHN BOYLAND. **The Interdependence of Effects and Uniqueness**. In *Workshop on Formal Techs. for Java Program*, 2001. 40

[62] DAVE CLARKE AND SOPHIA DROSSOPOULOU. **Ownership, encapsulation and the disjointness of type and effect**. *SIGPLAN Not.*, **37**(11):292–310, nov 2002. 40

[63] ROBERT L. BOCCHINO, VIKRAM S. ADVE, DANNY DIG, SARITA V. ADVE, STEPHEN HEUMANN, RAKESH KOMURAVELLI, JEFFREY OVERBEY, PATRICK SIMMONS, HYOJIN SUNG, AND MOHSEN VAKILIAN. **A Type and Effect System for Deterministic Parallel Java**. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 97–116, New York, NY, USA, 2009. Association for Computing Machinery. 40

[64] DOUGLAS E. HARMS AND BRUCE W. WEIDE. **Copying and Swapping: Influences on the Design of Reusable Software Components**. *IEEE Transactions on Software Engineering*, **17**(5):424–435, may 1991. 40

[65] HENRY G. BAKER. **"Use-once" variables and linear objects: storage management, reflection and multi-threading**. *SIGPLAN Not.*, **30**(1):45–52, jan 1995. 40

[66] TONY HOARE. **Null References: The Billion Dollar Mistake**. https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/, 2009. Date visited: 2024-09-16. 41

[67] PAULO SÉRGIO ALMEIDA. **Balloon types: Controlling sharing of state in data types**. In MEHMET AKŞIT AND SATOSHI MATSUOKA, editors, *Proceedings of ECOOP'97 – European Conference on Object-Oriented Programming*, pages 32–59, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. 41

[68] GUL AGHA. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986. 42

[69] NATHANAEL SCHÄRLI, STÉPHANE DUCASSE, OSCAR NIERSTRASZ, AND ANDREW P. BLACK. **Traits: Composable Units of Behaviour**. In LUCA CARDELLI, editor, *ECOOP 2003 – Object-Oriented Programming*, pages 248–274, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. 43

[70] ADELE GOLDBERG AND DAVID ROBSON. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1st edition, 1983. 44

[71] RANDAL L. SCHWARTZ, BRIAN D FOY, AND TOM PHOENIX. *Learning Perl*. O'Reilly Media, Inc., 6th edition, 2011. 44

[72] PYTHON SOFTWARE FOUNDATION. **Python 3.12.2 documentation**. `https://docs.python.org/3.12/`. 44

[73] SHU YU GUO, MICHAEL FICARRA, AND KEVIN GIBBONS. **ECMAScript 2023 Language Specification**. `https://ecma-international.org/publications-and-standards/standards/ecma-262/` 44

[74] L. Peter Deutsch and Allan M. Schiffman. **Efficient implementation of the smalltalk-80 system**. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, page 297–302, New York, NY, USA, 1984. ACM. 45

[75] Urs Hölzle, Craig Chambers, and David Ungar. **Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches**. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, page 21–38, Berlin, Heidelberg, 1991. Springer-Verlag. 45

[76] Jeremy Siek and Walid Taha. **Gradual Typing for Objects**. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, pages 2–27, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. 48

[77] The Hack and HHVM Team. **Hack Documentation**. https://docs.hhvm.com/hack/. Accessed: 2024-05-07. 48, 49

[78] The TypeScript Team. **TypeScript Documentation**. https://www.typescriptlang.org/docs/. Accessed: 2024-05-08. 48, 49, 70

[79] Jukka Lehtosalo and mypy contributors. **mypy 1.10.0 Documentation**. https://mypy.readthedocs.io/en/stable/. Accessed: 2024-05-28. 48, 72

[80] The Julia Project. **The Julia Language, V1.10.3**. https://raw.githubusercontent.com/JuliaLang/docs.julialang.org/assets/julia-1.10.3.pdf. 48, 49

[81] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. **Dynamic Typing in a Statically Typed Language**. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 213–227, New York, NY, USA, 1989. ACM. 48

[82] The Rust Reference Team. **The Rust Reference**. https://doc.rust-lang.org/reference/, 2023. Accessed: 2024-05-08. 48, 50, 54

[83] Elvis Pranskevichus and Yury Selivanov (editors). **What's New In Python 3.5**. https://docs.python.org/3.5/whatsnew/3.5.html, 2015. Accessed: 2024-04-23. 48, 56

[84] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. **PEP 484 – Type Hints**. https://peps.python.org/pep-0484/, 2014. Accessed: 2024-04-23. 48, 72

[85] The Pyre Team. **Pyre: A Performant Type-Checker for Python 3**. https://pyre-check.org/. Date visited: 2024-09-16. 48, 72

[86] The Pyright Team. **Pyright: Static Type Checker for Python**. https://microsoft.github.io/pyright/. Date visited: 2024-09-16. 49, 72

[87] Gavin Bierman, Martín Abadi, and Mads Torgersen. **Understanding TypeScript**. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 257–281, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. 49

[88] Gavin Bierman, Erik Meijer, and Mads Torgersen. **Adding dynamic types to C**. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP'10, page 76–100, Berlin, Heidelberg, 2010. Springer-Verlag. 50

[89] J.B. Wells. **Typability and Type Checking in System F are Equivalent and Undecidable**. *Annals of Pure and Applied Logic*, **98**(1):111–156, 1999. 50

[90] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., USA, 1978. 52, 53

[91] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. **Julia: A Fresh Approach to Numerical Computing**. *SIAM review*, **59**(1):65–98, 2017. 52, 53

[92] Konrad Zuse. *Der Plankalkül*. Berichte der Gesellschaft für Mathematik und Datenverarbeitung, 1972. (Manuscript prepared in 1945.). 52

[93] R.C. Holt and J.N.P. Hume. *Programming Standard Pascal*. RESTON PUBLISHING COMPANY, INC., A Prentice-Hall Company, Reston, Virginia, 1980. 53

[94] PHP Documentation Group. **PHP Manual**. https://www.php.net/manual/en/index.php. Date visited: 2024-09-16. 53

[95] The Go development team. **The Go Programming Language Specification**. `https://go.dev/ref/spec`, 2024. Accessed: 2024-04-30. 54

[96] The Go Team. **A Tour of Go: Slices**. https://go.dev/tour/moretypes/7. Accessed: 2024-09-22. 54

[97] Matt Austern. **Proposed Draft Technical Report on C++ Library Extensions**. Technical report, ISO/IEC PDTR 19768, 2005. 54

[98] Bjarne Stroustrup. *The C++ Programming Language, Fourth Edition*. Addison-Wesley, 2013. 54

[99] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. **The Java Language Specification, Java SE 8 Edition**, 2015. `https://docs.oracle.com/javase/specs/jls/se8/html/index.html`. 55

[100] The C Python Team. **CPython**. https://www.python.org/downloads/. Date visited: 2024-09-16. 56

[101] Sophie Kaleba, Octave Larose, Richard Jones, and Stefan Marr. **Who You Gonna Call: Analyzing the Run-Time Call-Site Behavior of Ruby Applications**. In *Proceedings of the 18th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2022, page 15–28, New York, NY, USA, 2022. Association for Computing Machinery. 71

[102] Zhifei Chen, Lin Chen, Yibiao Yang, Qiong Feng, Xuansong Li, and Wei Song. **Risky Dynamic Typing-related Practices in Python: An Empirical Study**. *ACM Trans. Softw. Eng. Methodol.*, **33**(6), jun 2024. 71

[103] Zhifei Chen, Yanhui Li, Bihuan Chen, Wanwangying Ma, Lin Chen, and Baowen Xu. **An Empirical Study on Dynamic Typing Related Practices in Python Systems**. In *2020 IEEE/ACM 28th International Conference on Program Comprehension (ICPC)*, pages 83–93, 2020. 71

[104] Ke Sun, Sheng Chen, Meng Wang, and Dan Hao. **What Types Are Needed for Typing Dynamic Objects? A Python-Based Empirical Study**. In Chung-Kil Hur, editor, *Programming Languages and Systems*, pages 24–45, Singapore, 2023. Springer Nature Singapore. 71

[105] Yi Yang, Ana Milanova, and Martin Hirzel. **Complex Python features in the wild**. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 282–293, New York, NY, USA, 2022. Association for Computing Machinery. 71

[106] Yun Peng, Yu Zhang, and Mingzhe Hu. **An Empirical Study for Common Language Features Used in Python Projects**. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 24–35, 2021.

[107] Zhifei Chen, Yanhui Li, Bihuan Chen, Wanwangying Ma, Lin Chen, and Baowen Xu. **An Empirical Study on Dynamic Typing Related Practices in Python Systems**. In *2020 IEEE/ACM 28th International Conference on Program Comprehension (ICPC)*, pages 83–93, 2020.

[108] XINMENG XIA, YANYAN YAN, XINCHENG HE, DI WU, LEI XU, AND BAOWEN XU. **An Empirical Study on the Impact of Python Dynamic Typing on the Project Maintenance**. *International Journal of Software Engineering and Knowledge Engineering*, **32**(05):745–768, 2022.

[109] B. WANG, C. LIN, W. MA, Z. CHEN, AND B. XU. **An empirical study on the impact of python dynamic features on change-proneness**. *International Conferences on Software Engineering and Knowledge Engineering*, 2015. 71

[110] TIOBE. **Index for September 2024**. https://www.tiobe.com/tiobe-index/. Date visited: 2024-09-22. 72

[111] THE TYPESCRIPT TEAM. **Type Compatibility**. https://www.typescriptlang.org/docs/handbook/type-compatibility.html. Accessed: 2024-09-22. 72

[112] COLLIN WINTER AND TONY LOWNDS. **PEP 3107 – Function Annotations**. https://peps.python.org/pep-3107/, 2006. Accessed: 2024-09-22. 72

[113] MOMINA RIZWAN, RICARDO CALDAS, CHRISTOPH REICHENBACH, AND MATTHIAS MAYR. **EzSkiROS: A Case Study on Embedded Robotics DSLs to Catch Bugs Early**. In *2023 IEEE/ACM 5th International Workshop on Robotics Software Engineering (RoSE)*, pages 61–68, 2023. 72

[114] FLEMMING NIELSON, HANNE R. NIELSON, AND CHRIS HANKIN. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 1999. 83

[115] CRISTIANO CALCAGNO, DINO DISTEFANO, JEREMY DUBREIL, DOMINIK GABI, PIETER HOOIMEIJER, MARTINO LUCA, PETER O'HEARN, IRENE PAPAKONSTANTINOU, JIM PURBRICK, AND DULMA RODRIGUEZ. **Moving Fast with Software Verification**. In KLAUS HAVELUND, GERARD HOLZMANN, AND RAJEEV JOSHI, editors, *NASA Formal Methods*, pages 3–11, Cham, 2015. Springer International Publishing. 83, 84

[116] JOHN C. REYNOLDS. **Separation Logic: A Logic for Shared Mutable Data Structures**. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, page 55–74, USA, 2002. IEEE Computer Society. 83

[117] JULIAN SEWARD AND ET AL. **Valgrind Documentation, Release 3.23.0**. https://valgrind.org/docs/manual/index.html, 2024. Accessed: 2024-09-22. 84