Technical Report

UCAM-CL-TR-902
ISSN 1476-2986

Number 902

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Programming language evolution

Raoul-Gabriel Urma

February 2017

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted September 2015 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Hughes Hall.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Programming languages are the way developers communicate with computers—just like natural languages let us communicate with one another. In both cases multiple languages have evolved. However, the programming language ecosystem changes at a much higher rate compared to natural languages. In fact, programming languages need to constantly evolve in response to user needs, hardware advances and research developments or they are otherwise displaced by newcomers. As a result, existing code written by developers may stop working with a newer language version. Consequently, developers need to search (analyse) and replace (refactor) code in their code bases to support new language versions.

Traditionally, tools have focused on the replace aspect (refactoring) to support developers evolving their code bases. This dissertation argues that developers also need machine support focused on the search aspect.

This dissertation starts by characterising factors driving programming language evolution based on external versus internal forces. Next, it introduces a classification of changes introduced by language designers that affect developers. It then contributes three techniques to support developers in analysing their code bases.

First, we show a source code query system based on graphs that express code queries at a mixture of syntax-tree, type, control-flow graph and data-flow levels. We demonstrate how this system can support developers and language designers in locating various code patterns relevant in evolution.

Second, we design an optional run-time type system for Python, that lets developers manually specify contracts to identify semantic incompatibilities between Python 2 and Python 3.

Third, recognising that existing codebases do not have such contracts, we describe a dynamic analysis to automatically generate them.

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Alan Mycroft, for his amazing support during my PhD. Alan has allowed me the freedom to explore my own ideas, and we even got to write a 400 page book together! I would also like to thank my PhD examiners David Greaves and Jeremy Singer for their constructive feedback.

My great thanks to Qualcomm and the Cambridge Computer Laboratory for offering me a scholarship to undertake this research.

I would also like to thank my friends and the Cambridge Programming Research Group for their continuous encouragements throughout the years.

Finally, I would like to thank my parents for their endless support and advice.

Contents

Chapter 1

Introduction

This dissertation has two related aims. First, it argues that programming language evolution is a complex and important subject which affects code maintained by developers. Second, it presents a search-based approach and three instances of tools to support developers in understanding the impact of programming language evolution in their codebases.

Programming language evolution Programming languages are the way developers communicate instructions to computers—just like natural languages let us communicate with one another. In both cases multiple languages have evolved. However, the programming language ecosystem changes at a much higher rate compared to natural languages. This dissertation claims that there are various reasons that cause this evolution. We argue that programming languages need to constantly *evolve* otherwise they are displaced by newcomers. In fact, programming languages need to change because of multiple pressures. We see two main forces influencing programming language evolution: *external* and *internal*. External forces include change in hardware, trends in the industry and research developments. For example, many traditionally object-oriented programming languages such as Java, C# and C++ have adopted functional programming features as a way to facilitate programming on multicore architectures. For instance, the most recent version of Java (Java 8) supports lambda expressions and a new declarative API to process collections.

Internal forces come from clumsiness or inefficiencies in existing versions of the language and may not necessarily be influenced by the ecosystem outside of the language. For example, language designers may need to fix bugs or design mistakes. In addition, developers may become frustrated with the verbosity of certain code patterns and practices. For example, iterating over a container before Java 5 required substantial set-up work. The for-each construct introduced in Java 5 addressed that specific issue.

Programming languages which do not react to these two forces are simply supplanted. Many examples of programming languages that were once popular include Ada, Algol, Cobol, Pascal, Delphi and Snobol.

Incompatible evolution Evolving programming languages is however challenging at various levels. Martin Buchholz, a JDK engineer, claims that “Every change is an incompatible change. A risk/benefit analysis is always required” [48].

Firstly, evaluating a proposed language change is difficult; *language designers* often lack the infrastructure to assess the change. Language designers often have to rely on mailing lists and community surveys. Recently, language designers of mainstream programming languages have argued for the use of real-world data to drive language evolution issues [63]. However, conducting studies can be difficult and time consuming. As a result, they are infrequently conducted and this may lead to arbitrary decisions. For example, older but rarely used features may remain in future language versions to maintain backward compatibility, increasing the language's complexity (e.g., FORTRAN 77 to Fortran 90). Similarly, developers have few means to understand the compatibility of their codebase with a new language version. How frequently do now-discouraged source-code patterns occur in their codebases?

Secondly, the impact on *developers* can be negative. For example, if two language versions are incompatible developers may reject the new language version. However, older versions are often not supported by language designers. Consequently, developers may miss important bug fixes and security updates. This is a major industrial problem. For instance, Python 2 and 3 have major incompatibilities. This issue affects large companies such as Google that maintain millions of line of code written in Python 2. Alternatively to rejecting a new language version, developers may choose to co-evolve their codebase (which may be costly) but what guarantees do they have that their code still behaves as intended after migrating to a newer version?

Every change to a programming language has the potential to affect programs written in this language. A recent study shows that developers lack awareness of the subsequent possible incompatibilities [52]. To tackle this issue, this dissertation builds upon previous work and establishes a classification of incompatible changes composed of six categories by analysing bug reports and compatibility guides from several programming languages.

Search-based approach Language designers and developers have few means to understand the impact of programming language evolution. An established research field in software engineering is called *change-impact analysis* [34, 112, 109]. This is a process that predicts and determines the parts of a software system that can be affected by changes to that same system. It useful to software maintainers for several reasons. For example, it lets them estimate the cost of changes. As a result, they can plan further development and testing resources accordingly. This dissertation relates the idea of change impact analysis to programming language evolution. We argue that developers also need tools so they can determine the parts of their codebases which are affected by changes in a programming language. More concretely, we present a search-based approach summarised in Figure 1.1. Concretely, we develop three tools that language designers and developers can make use of:

1. *Search*: a scalable and expressive source code query language to locate code patterns using full source-code detail in large codebases. It can be used by language designers to rapidly test programming language evolution hypotheses such as "Are covariant arrays used frequently by developers?". In addition, it can be used by developers to understand the compatibility of their codebase with a new language version by searching for deprecated patterns or source of incompatibilities.

2. *Verify*: a runtime type checking system exemplified for Python as a form of *migration contracts* based on function type signatures that easily integrates in existing developer infrastructure. Developers can specify contracts and get a clear diagnosis

and explanation if type contracts are violated by migrating to a different language version.

3. *Infer*: A dynamic light-weight type inference tool for Python to automatically provide insights useful for migration about a Python program. In addition, it alleviates the burden of manual annotation of function type signatures which are used for verification.

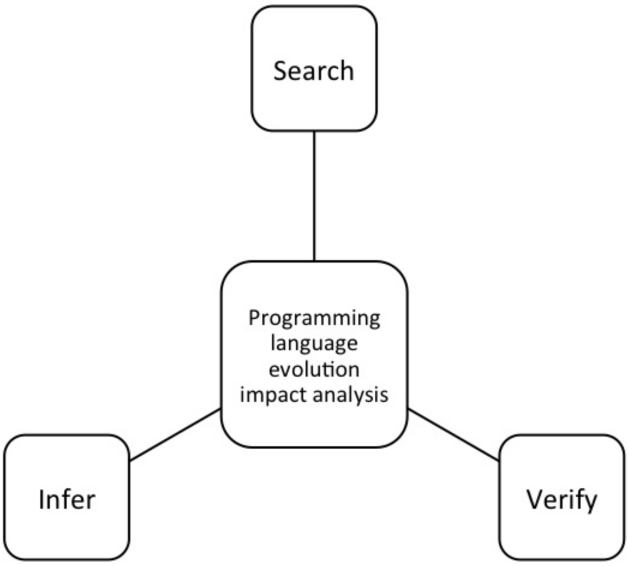


Figure 1.1: Search-based approach to programming language evolution impact analysis

Our results suggest that search-based tools can provide useful insights to developers to manage programming language evolution. We found that the techniques we developed are lightweight and simple to integrate in large programs and can efficiently diagnose programming language evolution issues that would be difficult for developers to manually locate and understand.

1.1 Chapter outline

The rest of this dissertation is structured in the following manner.

Programming language evolution (Chapter 2) characterises factors driving language evolution based on external versus internal forces. It is based on a study of language feature specifications from several programming languages. Some material of this chapter was published in the book Java 8 in Action [125].

Notions of language compatibility (Chapter 3) establishes a classification of incompatible changes introduced into programming languages that affect developers. It provides a survey of use cases based on an analysis of compatibility guides and bug reports from several programming languages.

Expressive and scalable source code querying (Chapter 4) addresses the problem of searching for complex code patterns to help developers and language designers examine code to understand the impact of language changes. This work was published in Science of Computer Programming [127].

Run-time type-checking for Python (Chapter 5) addresses the problem that certain classes of incompatibilities are difficult to locate accurately in dynamically-typed programming languages. It contributes a run-time type-checking system for Python which let programmers annotate their code to produce diagnosis and explanations about the location of these incompatibilities. This work influenced the design of a standard type annotation language as part of a Python enhancement proposal by Rossum et al [132].

Dynamic inference of Python function signatures (Chapter 6) contributes a dynamic analysis to automatically provide migration insights about a Python program. This analysis generates function type signatures to alleviate the burden of manual annotation for programmers.

Conclusions (Chapter 7) concludes by summarising the contributions made in this dissertation and discusses ideas for further work in the field of programming language evolution.

Chapter 2

Programming Language Evolution

In this chapter, we argue that programming language evolution is an important subject which affects developers, language designers and researchers. Specifically, we investigate the driving forces for changes introduced in programming languages.

Subsequent chapters explore various practical aspects of programming language evolution. In particular, we focus on dealing with incompatible changes. For now, we consider programming language evolution in general.

We start our exposition by investigating the programming language ecosystem history and relate it to Darwinian theory of "*survival of the fittest*". In addition, we link programming language evolution to the field of evolutionary linguistics.

Next, we establish a framework to frame programming language evolution by analysing programming-language specifications and their associated migration guides. Our framework is based on *external forces* such as research developments and industry trends and *internal forces* such as how developers are using the programming language in practice. We focus our survey on the evolution of Java, C# and Python as microcosms because of their popularity. However, we argue that the principles apply to other programming languages too.

Next, we claim that today's programming language ecosystem has become heavily controlled by large technology firms. We discuss the business-driven motivations for them wanting to more tightly control programming language evolution. An example of this which we describe is the release of new programming languages such as Swift, Go and Hack owned by Apple, Google and Facebook respectively for their internal use as well as development on their platforms.

Finally, we summarise major research strands in the context of programming language evolution as well as positioning the contributions of this dissertation.

2.1 Darwinian evolution

With the 1960s came the quest for the perfect programming language. In a landmark article, Peter Landin noted in 1966 that there had already been 700 programming languages and speculated on what the next 700 would be like [77]. Many thousands of programming languages later, academics have concluded that programming languages behave like an ecosystem: new languages appear and old languages are supplanted unless they evolve. We all hope for a perfect universal language, but in reality certain languages are better fitted for certain niches [51]. For example, C and C++ remain popular for building operating systems and various other embedded systems, because of their small runtime footprint and in spite of their lack of programming safety. This lack of safety can lead to programs crashing unpredictably and exposing security holes for viruses and the like; indeed, type-safe languages such as Java and C# have supplanted C and C++ from various applications when the additional runtime footprint is acceptable. Prior occupancy of a niche tends to discourage competitors. Changing to a new language and tool chain is often too painful for just a single feature, but newcomers will eventually displace existing languages, unless they evolve fast enough to keep up. For example, once popular languages such as Ada, Algol, COBOL, Pascal, Delphi, and SNOBOL have now faded.

We see this evolution as a Darwinian evolution: programming languages need to evolve to survive. Knuth et al. already observed in 1976 that programming languages need to evolve rapidly in response to current needs or are displaced [74]. In his paper, he surveys the evolution of high-level languages from 1945 to 1957 and their ability to express algorithms. He compares these languages by implementing an algorithm called TPK designed to highlight the power of each of them. He notes that languages up to Fortran 1 did not support features that are now seen as fundamental in computer science such as data structures other than arrays, control structures and recursion. Consequently, such languages did not manage to have a larger impact. Social factors such developer demographics and social networks are also an important driver of programming language evolution; this is often overlooked when considering the adoption of new features [85].

Similar evolutionary pressures are highlighted in research fields outside of computer science, for example, in evolutionary linguistics (a field dedicated to researching the origins and development of natural languages such as English [47]). As with the deprecation of programming languages, many once-popular languages are now rarely spoken. However, natural languages evolve over a scale of decades and centuries. This allows society to adapt gradually. Programming languages evolve at a much faster rate, on the scale of years, which puts developers under greater pressure to keep up.

2.2 External vs. internal forces

In this section, we propose a framework based on two distinct forces that are driving programming language evolution:

1. **External forces** such as changes in hardware, industrial trends and research developments. For example, the ubiquity of multi-core architectures has forced various programming languages such as Java to include simpler programming language constructs to help programmers efficiently work with parallelism.

2. **Internal forces** such as inefficiencies in the language or perceived clumsiness. For example, the use of certain language features may prove to be error-prone or inefficient. Consequently, language designers need to address this to satisfy developers that are working using their programming languages on a day to day. As example of that is implementing the enumerator pattern in older versions of C#, which required a lot of set-up work and was simplified in C# 2.0. In addition, bugs or ambiguities in the language specification implementation that need to be fixed are also an internal force. For example, several languages have bug reports available online that the community can contribute to by reporting problems they encounter when working with the programming language.

We argue that the evolution of existing mainstream programming languages can be characterised into two main categories: the evolution of languages tends to be either a combination of both external and internal forces or just internal forces. By contrast, new but not yet popular (or indeed novel) programming languages are mostly driven from external forces by their nature of being new languages. Figure 2.1 visualises this through a two-axis graph representing external and internal forces. The area (1) represents changes introduced principally by internal forces. The area (2) represents changes that are driven from both external and internal forces. These are two main areas representing the evolution of mainstream programming languages. The area (3) represents changes driven principally by external forces and in general corresponds to non-mainstream programming languages.

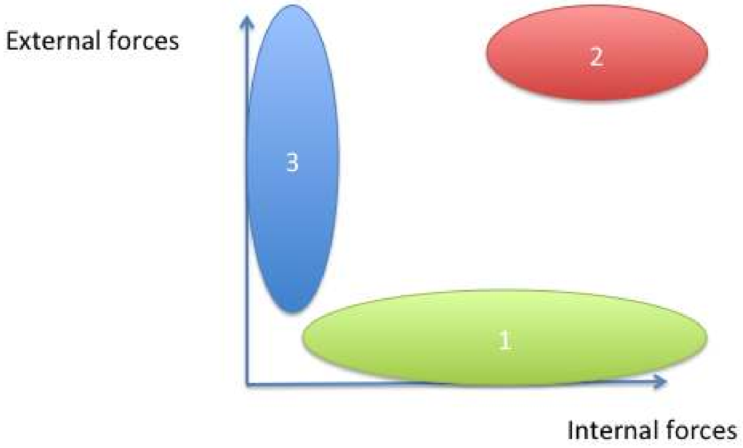


Figure 2.1: External vs. Internal forces

In the next section, we present a survey of various language features to illustrate how external and internal forces interact in the context of programming language evolution.

2.3 Survey of language features

In the following section, we discuss the insights of a survey of language-feature proposals and specifications from Java, C# and Python within our framework of external and internal forces. Table 2.1 summarises the survey.

Programming Language	Version	Feature	Forces
Java	1.8	Streams	External + partly internal
	1.8	Lambda expressions	External + partly internal
	1.7	Improved type inference for generic instance creation	Internal
	1.7	Improved Exception Handling	Internal
	5	Generics	External + partly internal
	5	Enhanced for loop	Internal + partly external
C#	3.5	LINQ	External + partly internal
	2.0	Generators	Internal + partly external
Python	2.5	Conditional expression	Internal
	2.0	List comprehensions	Internal + partly external

Table 2.1: Survey summary

2.3.1 Java

In this subsection, we discuss the introduction of various major and minor changes in different versions of Java by reviewing their specifications. Specifically, we consider:

- Lambda expressions and the Streams API introduced in Java 8
- Improved type inference for generic instance creation and improved exception handling in Java 7
- Generics and the enhanced for loop introduced in Java 5

Java 8 streams

Programmers are increasingly dealing with large sets of data (datasets of terabytes and up) and wishing to exploit multi-core computers or computing clusters effectively to process it. Java's existing Collection API makes this difficult to do efficiently and correctly without expert knowledge. To support this need Java 8 introduced the Streams API which is a lazily evaluated collection-like data structure which can execute its operations in parallel. For example, naively calculating the variance of a data set in parallel can now be expressed in a few lines compared to equivalent code using the lower-level fork/join framework [124]:

```
public static double varianceStreams(double[] population){
   double average =
       Arrays.stream(population).parallel().average().orElse(0.0);
   double variance =
       Arrays.stream(population).parallel()
             .map(p -> (p - average) * (p - average))
             .sum() / population.length;
   return variance;
}
```

We see the introduction of streams as driven by two external forces. On the one hand, multi-core architectures are now mainstream. To get additional performance, programmers have no choice but deal with these architectures. As a result, language designers need to provide new means for programmers. On the other hand, the addition of Streams to Java was influenced by the existence of similar offerings outside of Java which provides an additional external force. In fact, the original Java proposal explicitly says "Linq and Plinq in particular are seen as extremely valuable by .NET developers and a subject of much envy by Java developers" [14]. One can note that programming for multi-core has always been very error-prone for programmers [119]. The introduction of a higher-level API can therefore be seen as also partly driven from internal forces.

Java 8 lambda expressions

Java 8 also introduced closures. This introduction is seen as an external force by several languages offering this feature. In fact, the original Java 8 proposal states [15]:

> "Many other commonly used object-oriented programming languages, including those hosted on the JVM (for example Groovy, Ruby, and Scala) and hosted on other virtual machines (C# on the CLR), include support for closures. Therefore, Java programmers are increasingly familiar with the language feature and the programming models it enables."

However, the main motivation given there is to "simplify the creation and consumption of more abstract, higher-performance libraries". The idea of representing a function as an object could be implemented in earlier versions of Java via various design patterns including anonymous inner classes. However, these are verbose solutions which were unsatisfactory for programmers. The introduction of lambda expression can therefore also be seen as driven from internal forces because it is syntactically simpler than an equivalent anonymous inner class. Compare how to create a predicate for testing whether an apple is green before Java 8:

```
Predicate<Apple> isGreenApples = new Predicate<Apple>(){
    public boolean test(Apple a){
        return a.getColour() == GREEN;
    }
};
```

and using Java 8:

```
Predicate<Apple> isGreenApple =
    (Apple a) -> a.getColour() == GREEN;
```

Java 7 improved type inference for generic instance creation

Java 7 introduced a form of type inference for generic instance creation. Instead of writing:

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

Programmers can now write more concisely:

```
Map<String, List<String>> myMap = new HashMap<>();
```

The `<>` (called the *diamond operator*) infers the type arguments from the context. Consequently, the code is less verbose by avoiding the duplication of writing complex types. This feature was originally proposed by the community on a mailing list [22]. The author of the proposal highlights that there are alternative available ways to express this idea but which are considered bad styles: "The requirement that type parameters be duplicated unnecessarily like this encourages an unfortunate overabundance of static factory methods, simply because type inference works on method invocations (and doesn't for `new`)". This feature is therefore mainly driven from an internal force whereby a lacuna in the language encouraged developers to use clumsier alternatives.

Java 7 improved exception handling

Java 7 introduced two enhancements to exception handling. First, a single catch clause can now catch more than one exception type which lets developers write more concise code. For example:

```
catch (IOException ex) {
    logger.log(ex);
    throw ex;
}
catch (SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

can be written as:

```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

This enhancement is driven from an internal force. Indeed, developers were writing and maintaining duplicate code for a simple pattern. The addition of multi-catch results in simpler code.

Second, Java 7 corrected an issue with how exceptions are rethrown. The code below illustrates the changes. Previously, the statement `throw exception` would throw a `Foo` exception. Now, it throws a `DaughterOfFoo` exception.

```
class Foo extends Exception {}
class SonOfFoo extends Foo {}
class DaughterOfFoo extends Foo {}

class Test {
    void test() {
        try {
            throw new DaughterOfFoo();
        } catch (final Foo exception) {
            try {
                throw exception; // first incompatibility
            } catch (SonOfFoo anotherException) {
                // second incompatibility: is this block reachable?
            }
        }
    }
}
```

Prior to Java 7, a rethrown exception was treated as throwing the type of the `catch` parameter. Now, when a `catch` parameter is declared `final`, the type is only known to be the exception types which are thrown in the `try` block and which also are a subtype of the `catch` parameter type. This enhancement is driven purely from an internal force: fixing a shortcoming in the language. This improvement led to two incompatibilities where existing code behaved differently.

Java 5 generics

Java 5 introduced generics, a feature which adds additional compile-time type safety checking. It removes the needs for explicit casts from `Object` which reduces the possibility of runtime type errors [41]. For example, prior to Java 5 programmers had to write code such as:

```
List apples = new ArrayList();
apples.add(new Apple());
Apple a = (Apple) apples.get(0);
```

The `List` declares a method `get` to have the return type `Object`. Consequently, an explicit cast is required to tell the rest of the program that the elements are of type `Apple`. If the programmer gets the cast wrong, a runtime-error occurs. This can be frustrating for programmers and represented an internal force pushing for change. With generics, programmers can now write the following more concise and safer code:

```
List<Apple> apples = new ArrayList<Apple>();
apples.add(new Apple());
Apple a = apples.get(0);
```

Clearly the motivation of this feature is to help developers write safer code. However, the generics proposal also included polymorphic methods and the ability to specify user-defined generics. Parnin et al. recently showed that generics have suffered from slow programmer uptake apart from simple use of homogenous collections; this weakens the internal force justification [101]. In addition, the implementation of generics for Java 5 was based on external research work by Bracha et al. [42], which detailed a fully backwards compatible proposal for Java. They noted that many other languages at the time supported generics: “Adding generics to an existing language is almost routine”. This supports the argument that the addition of generics was mostly driven from external forces.

Java 5 enhanced for loop

Prior to Java 5, iterating over a collection was verbose. Programmers had to explicitly use the methods of an Iterator object. For example, the following code iterates through a list of numbers:

```
Iterator itr = numbers.iterator();
while (itr.hasNext()) {
    Object e = itr.next();
    System.out.print(e + " ");
}
```

Programmers can now use the enhanced for loop (i.e. for-each) which reduces verbosity by hiding the boilerplate:

```
for (Integer e: numbers) {
    System.out.println(e + " ");
}
```

The initial feature proposal highlights the internal force for this construct: “ease of development and significantly reduces verbosity of a very common construct” [16]. In addition, the official guide strongly recommends its use “So when should you use the for-each loop? Any time you can. It really beautifies your code” [9]. The for-each construct was clearly introduced to solve a programming clumsiness. Note that other competitor mainstream languages such as C# already supported a similar construct. This may have acted as an external force for the language designers to include it in Java.

2.3.2 C#

In this section, we review the introduction of two features in C#:

- LINQ in C# 3.5
- Generators in C# 2.0

LINQ

C# version 3.5 added the embedded query language LINQ [82], which lets programmers process data in declarative manner. Before the introduction of LINQ, programmers had to work with multiple different languages (e.g. SQL for database and XPath for XML) or APIs to query data. Using LINQ, programmers can now write the following uniform query where the data could come from various sources such as a database, an in-memory collection or an XML file:

```
var results =
     data.Where(invoice => invoice.Price > 5000)
         .Select(invoice => {invoice.ID, invoice.Customer});
```

The popularity of databases and the need to query them is the external force that motivated LINQ. Indeed, nearly every application needed to work with data. In addition, programmers tended to adopt the declarative style of SQL. However, the introduction of LINQ can also be seen as an internal force since existing mechanisms in the language, such as the existence of multiple APIs for various data formats, were unsatisfactory and time-consuming to work with.

Generators

C# lets programmers define iterators (called enumerators) associated with a user-defined type. Programmers can then use the `foreach` construct to iterate over it as discussed earlier. However, this process requires a lot of boilerplate code and is known as implementing the *enumerator pattern* using an external class. C# 2.0 introduced an improvement that enable programmers to *create* iterators as generators. This feature can make it possible for classes to declare how the `foreach` statement will iterate over their elements. For example, instead of declaring and implementing a separate `IEnumerator` which needs to manage the state of the iteration, C# 2.0 programmers can now directly implement the `GetEnumerator` method using a `yield` statement as follows:

```
public class Range : IEnumerable {
    private int start;
    private int end;

    public Range(int s, int e) {
        start = s;
        end = e;
    }

    public IEnumerator GetEnumerator() {
       for(int i = start; i <= end; i++) {
          yield return i;
       }
    }
}
```

This feature is driven by an internal force: the enumerator pattern is difficult and time-consuming to implement and programmers need to do it frequently. The proposal for the iterator feature takes inspiration from research languages such as CLU, Sather and Icon, which supported a similar feature. This is an external force that influenced the introduction of this feature into C# [3].

2.3.3 Python

In this section we review the introduction of two features in Python:

- Conditional expressions in Python 2.5
- List comprehensions in Python 2.0

Conditional expressions

Python 2.5 added conditional expressions with syntactic form of `X if C else Y`. The initial proposal argues that programmers were making several errors when attempting to achieve the same affect using the `and` and `or` operators [131]; this motivated the need for a new construct. As a result, conditional expressions can be seen as driven by an internal force.

List comprehensions

Python 2.0 added list comprehensions to provide a more concise way to create lists in certain situations. Furthermore, list comprehensions was introduced as a concise replacement for inefficient combinations of `map()`, `filter()` and nested loops. This is clearly an internal force. For example, the following code:

```
map(lambda x: x**2, filter(isprime, range(1, 99))
```

is equivalent to the following Python 2.0 code using list comprehensions:

```
[x**2 for x in range(1, 99) if isprime(x)]
```

However, the mailing list discussing the proposal mentions that the syntax took inspiration from Haskell. This represents an external force.

2.3.4 Discussion

It is interesting to notice that smaller additions to features in mainstream languages are mainly driven from internal forces. However, subsequent features are driven by both external and internal forces. Specifically, external forces are driving the early development of the programming language ecosystem with the creation of new features in nicher programming languages. These features are then only later incorporated as evolutionary changes into mainstream languages. Another force which we have not considered is that of business-driven evolution, which we merit a section to itself.

2.4 Business-driven evolution

So far we have argued that programming language evolution is driven based on external and internal forces. We noted that "industrial need" is an important external force. However, we also believe that an emerging related actor which influences the programming language ecosystem and its evolution are large technology firms such as Apple, Google and Facebook.

We develop three arguments as to why large technology companies are becoming new influences in programming language evolution:

1. Control of the ecosystem
2. IP protection
3. Revenue

2.4.1 Control of the ecosystem

In the last decade, a new phenomenon has hit us. Many large technology companies now control existing programming languages or create their own programming language. For example, Apple recently released Swift, a new programming language for iOS and OS X. Google released Go, a language for systems programming. Google also released Dart, a programming language to replace Javascript. Facebook recently released Hack, a programming language that can be seen as a new version of PHP. Red Hat released Ceylon, a programming language born out of "frustration with Java" and designed to replace it [4]. We see this strategy as a way to isolate themselves against "natural" programming language evolution.

One can see the release of these industrially supported languages as a good cause for the community. However, there is also a business-driven motivation. In fact, it is attractive for a company to minimise dependency on external technology which may change in directions that are inconvenient to their business.

As noted above, Facebook recently introduced Hack. The programming language was born because of internal forces: programmers at Facebook were frustrated with shortcomings with PHP and not being able to address them. As a result, they created a similar programming language but more tightly specified which they can control. In addition, it can also benefit recruitment of skilled engineers which see programming in PHP as unattractive. Moreover, Facebook has thousands of engineers which use the language at work that can then act as evangelists; this acts as a counter-force to the natural evolution of PHP.

With the exploding growth of web applications, Google released Dart, a programming language which aims to replace the ubiquitous JavaScript. In addition, Google is working on integrating Dart inside Chrome, the most popular browser at the moment, which is also developed by Google. As a consequence, Dart may steal developers from the JavaScript community. Microsoft recently released Typescript also an alternative to ECMAScript and seen as a direct competitor to Dart.

A related issue is that of several companies that hire the main leader of a language community. For example, Dropbox has hired Guido van Rossum, the inventor of Python, because they make heavy use of Python internally. Facebook has hired Andrei Alexandrescu, the language designer of the programming language D. These kinds of hire exposes interesting conflict-of-interest questions about the future direction of programming language evolution. Is it driven entirely by the community or is it influenced by business needs of the company that employs a key developer?

2.4.2 IP protection

Oracle recently acquired Sun. Through this acquisition they have taken ownership of Java as well. This can be seen as a strategic move to control a programming language that Oracle's competitors such as Google and IBM heavily rely on. For example, Google has invested its future in Java by making it the main language on the Android platform. Relatedly, Oracle recently sued Google for copyright infringement of the Java API [20]. In fact, Google's implementation of its mobile operating system Android used the same names, structure and functionalities as the Oracle Java API to attract developers in the Java community to develop on their platform.

2.4.3 Revenue

Another reason why it is attractive to control a programming language is that it can result in additional revenue. For example, Jetbrains, a company behind the popular IntelliJ IDEA IDE, is developing Kotlin, an alternative language to Java and Scala. Jetbrains openly says that "we expect Kotlin to drive the sales of IntelliJ IDEA" [28].

We can see a similar historical example with Pascal. The company Borland developed two dialects to Pascal, which shipped with a paid integrated development as apart of their main business model: Turbo Pascal (cheaper) and Borland Pascal (more expensive).

However, programming languages can bring revenue from multiple different avenues. For example, Google's programming languages are offered as part of their cloud-computing offerings. By building a large community of users Google is able to sell more of their products. Microsoft provides paid tools, support and training to develop using C#. Typesafe, the company co-founded by Martin Odersky, the creator of Scala, influences the development of Scala. Its main business model is to offer a package that includes commercial support, maintenance, and tools to develop in Scala. Typesafe also offers training and consulting services on Scala frameworks they develop.

2.4.4 Discussion

The conclusion from this section is that programming language evolution is more subtle than we can think. As we discussed, there are several forces which can influence the programming language ecosystem. However, large technology firms is also one relevant force which may not always act in the best interest of "natural" programming language evolution with developer needs in mind.

2.5 Dealing with programming language evolution

We discussed that programming language evolution may happen in response to several factors. Changes to the language raise the question of how can developers deal with existing code bases? In fact, as programming languages evolve, older versions are often discontinued and not supported any longer. In addition, new features in languages often let programmers express intent more clearly compared to older idioms. It is possible for programmers to manually migrate codebases to a newer version. However, manual location and transformations of relevant source code in a program are time-consuming and prone to errors. This is even more pronounced for backward incompatible changes introduced in the context of programming language evolution as we discuss in the rest of this dissertation.

Previous work has suggested the use of automated techniques to support developers with programming language evolution. For instance, Pirkelbauer et al. presented the notion of *source code rejuvenation*: the automated migration of legacy code [107, 106]. They contrast it to the notion of *refactoring* in several ways. While refactoring involves improving the design of existing code in a behaviour-preserving manner, they argue that source code rejuvenation is a process to improve existing code by taking advantage of improvements to a new version of the language and associated libraries (e.g. eliminating outdated idioms). In contrast with refactoring, source code rejunevation does not necessarily have to preserve behaviour. Various researchers have released automated tools to support source code rejunevation and argued that it benefits maintenance, verification, extension, and understandability of code. For example, Overbey et al. presented the tool Photran to replace use of old Fortran features with their modern equivalents [98, 99]. This work was extended by Orchard et al. who developed CamFort to show that it is possible to locate and transform sophisticated programming patterns in Fortran using a general analysis infrastructure [97].

However, automated migration may not always possible. A new version of a programming language may introduce incompatibilities with respect to an older version. These incompatibilities can be time-consuming and expensive for programmers to locate in large code bases. In addition, incompatibilities can also be subtle and difficult to understand as a developer. This problem is the focus of this dissertation. We argue that automated support for searching code bases for incompatibilities is also necessary to support developers with programming language evolution. The next chapter introduces notions for language compatibility and the remaining chapters contributes three techniques to support developers in searching and analysing their code bases.

Chapter 3

Notions of language compatibility

In the previous chapter, we presented an analysis of the causes of programming language evolution. We argued that changes introduced in programming languages are driven by both external and internal forces.

This chapter focuses on changes that are not compatible with previous versions of a programming language. Subsequent chapters deal with tools to support developers in locating changes that are not compatible.

A compatible change is a change that keeps an existing program working as it did before. A change that is incompatible may produce a different runtime behaviour, or the source code of the program may not compile using the new language compiler. We refer to these compatible changes in a new language version as *backward-compatible* changes. Figure 3.1 formalises this notion. We define backward compatibility between two versions `v` and `v'` as a predicate which indicates that all the programs in the syntax set of version `v` have the same behaviour in version `v'`[1].

$Bcomp(v, v') = \forall P \in Lang(v)\ we\ have\ Behaviour_v(P) = Behaviour_{v'}(P)$

$Lang(v)$: syntax set for language version `v`
$Behaviour_v(P)$: Behaviour of program `P` with language version `v`

Figure 3.1: Forward and Backward Compatibility

A recent study shows that programmers lack awareness of compatibilities beyond compile errors [52]; this forms part of the motivation for this chapter. A naive definition of a compatible language change from version 1 to version 2 is a change where every program in language version 1 is a valid program in version 2 and has the same behaviour.

This chapters has two aims. First, we argue that language compatibility is more subtle and goes beyond this simplistic definition. Second, the main contribution of this chapter is to establish an extended non-exhaustive classification of backward-compatible changes. We build upon previous work [44, 48] which introduced *behavioural compatibility*, *source compatibility* and *binary compatibility*. We constructed this extended classification by analysing bug reports and compatibility guides from several languages including Java, Python and PHP.

Concretely, we make two advances:

[1] Note that forward compatibility refers to a *design principle* that allows a system to accept input intended for a later version rather than a *property* of a system

1. We further include *performance-model compatibility*, *data compatibility* and *security compatibility*.

2. We present a detailed composition of the different kinds of behavioural compatibility, source compatibility and binary compatibility.

We use this classification in later chapters to design tools to support developers in searching the location of changes that are not backward compatible. One goal of this classification is to provide programmers with a checklist to better understand the possible compatibility issues when migrating their codebases to different language and library versions. Figure 3.2 summarises the classification visually.

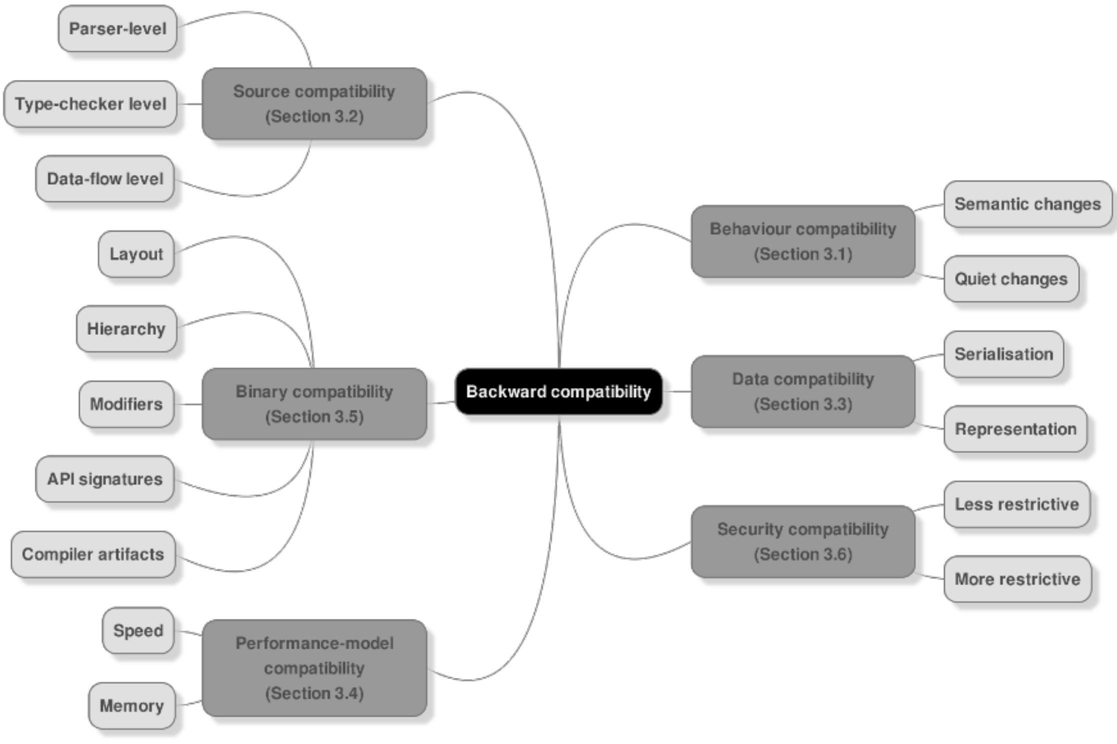


Figure 3.2: Six categories of backward compatibility

Figure 3.2 shows that backward compatibility is a topic that is more complex than one immediately imagines. In fact, the figure shows six main categories of backward compatibility: source, binary, performance-model, behavioural, data and security. Each category is then further divided with finer-grained sub-categories. The rest of this chapter investigates each category in details with real-world use cases of incompatible changes introduced in programming languages. We also discuss that incompatible changes do not always fit into a single category but can have cross-cutting characteristics.

3.1 Behavioural compatibility

Behavioural compatibility means that a program written in version `n` of programming language will produce exactly the same behaviour at runtime given the same input using version `n + 1` of the programming language. In this section, we classify behaviour incompatible-changes into *semantic* and *quiet* changes.

3.1.1 Explicit semantic change

A change may alter the meaning of a program in a different version of the programming language. We highlight two such semantic changes exemplified in real programming languages: producing a different value or result and using a different type coercion.

Different result In Python 2 an expression such as `1/2` returns `0` (of type int) as both arguments of the division operator are ints. However, `1.0 / 2` returns `0.5` as one of the argument is a `float`. To contrast, in Python 3 the division operator always returns a `float` and the result is `0.5` in both cases.

As another example, the function `ip2long()` in PHP 4 returns an integer representation of an IP address given its string representation or -1 if it fails. In PHP 5 it now returns FALSE when an ill-formed IP address is passed as argument to the function. These examples can result in behavioural incompatibilities if the control flow of the program is based on the result of that function.

Such semantic changes can also be the result of fixing an implementation bug. For example, prior to Java 8, calling `BigDecimal.stripTrailingZeros` on a string value numerically equal to zero would return that value (e.g. 0.00 would return 0.00 instead of 0). Now the method instead returns the constant `BigDecimal.ZERO` [90].

Different coercion In PHP 4 an object with no class member variables is considered empty. However, in PHP 5 it is no longer considered empty. The code below in PHP 5 would print "bar" but in PHP 4 would not print anything.

```
<?php
class foo { }
$o = new foo();

if ($o) {
    echo "bar";
}
?>
```

3.1.2 Quiet semantic changes

The ANSI/ISO C rationale document formalises the notion of *quiet changes* [69]:

> "Any change to widespread practice altering the meaning of existing code causes problems. Changes that cause code to be so ill-formed as to require diagnostic messages are at least easy to detect. As much as seemed possible consistent with its other goals, the Committee has avoided changes that quietly alter one valid program to another with different semantics, that cause a working program to work differently without notice. In important places where this principle is violated, the Rationale points out a *quiet change*".

Such changes are not specific to C and can occur in other programming languages too. We highlight four categories of quiet changes exemplified in Java: unspecified behaviour, improper behaviour, timing and threading model.

Unspecified behaviour In Java 7, build 129, the implementation of the methods `getMethods`, `getDeclaredMethods` and `getDeclaredConstructors` from `java.lang.Class` changed the order in which they return the methods and constructors of a class. As a result, user code that relied on this order might not work as they did before [17]. However, the specification for these methods never prescribed a particular order. But some users assumed there was an order based on previous implementation behaviour. Such user assumptions in the presence of "underspecified specifications" can be a major source of quiet semantic changes.

Improper behaviour Java 1.2.0's reflection API let users change the value of a final field using the `Field.setAccessible` and `Field.set` method. However, this was considered as a bug in the implementation rather than the specification and was prohibited in future versions by throwing an exception [88]. Consequently, any programs that relied on this pattern prior to Java 1.2.0 would unexpectedly terminate in later releases.

Timing assumptions One can also imagine timing assumptions made by users can also result in quiet changes. For example, the control flow of a program may depend on the observed time of a built-in operation (e.g. a time out mechanism). If the implementation of the operation is changed or even run on different hardware, the program may not behave as expected. Related is the idea of timing attack [75]. An attacker may be able to compromise an encryption algorithm by analysing the time taken to encrypt different inputs. For example, Java 6u17 updated the implementation of `MessageDigest.isEqual` which was vulnerable to a timing attack because it was doing byte-by-byte comparison and returning on first inequality [5]. It was modified to run in constant time instead.

Threading model Assumptions about the threading model could result in errors. For example, changing a method from `synchronized` (prevents threads interference) to non-`synchronized` in Java could result in race conditions over shared data that can affect the behaviour of the running program. As another example of how threading model assumptions can affect the outcome of a program, in the previous chapter we discussed the Streams API in Java 8 which lets developer express data-processing queries that abstract the underlying threading model. The following code will produce a different output depending on whether a sequential or parallel stream is passed as argument to the method `totalCost` because the `DELIVERY_FEE` could be added multiple times by more than one thread:

```
int totalCost(Stream<Purchase> items) {
  return items.reduce(DELIVERY_FEE,
                      (tally, item) -> tally + item.cost());
}
```

3.1.3 Discussion

We have described two classes of behavioural imcompatibilities: *explicit* semantic changes which violate an existing specification and *quiet* semantic changes which introduce semantic changes but do not necessarily break a previous specification.

An interesting observation is that behavioural changes can also be classified in terms of “visibility” for programmers. For example, a function producing a different value (0.5 instead of 0) can be inspected in the source code by identifying the uses of that function. To contrast, a subtle rule change in type coercion (e.g. PHP 4’s empty definition) is insidious for programmers because it is difficult to immediately observe or understand by simply inspecting the source code. In fact, conversions are typically implicit (no syntactic keywords), which renders them invisible to the programmer.

In addition, one can notice that changes do not necessarily have a clear-cut categorisation. In fact, change may have cross-cutting characteristics. For example, we discussed that changes in timing assumptions of an operation can result in behavioural incompatibilities. Moreover, it may break the performance-model expected by the user. Furthermore, changes in timing assumptions can also result in security problems via timing attacks as we discussed earlier.

3.2 Source compatibility

In its simplest form, source compatibility means that an existing program will still compile after introducing a change. For example, adding a method to an interface in Java is not source compatible. Indeed, existing implementations will not re-compile because they first need to implement the new method. This incompatibility occurs at the type-checker level. In this subsection, we make two statements. First, we argue that source-incompatible changes can be sophisticated and go beyond simple errors such as adding a new reserved keyword in the language. Second, we show that most source incompatibilities can be categorised into three main categories: parser, type-checker and data-flow.

3.2.1 Lexer/Parser level

Source incompatibilities can occur when language designers alter the grammar rules of the language such as introducing new reserved keywords.

Lexical For example, Java 1.4 introduced a new `assert` keyword. As a consequence, existing programs that used `assert` as an identifier became invalid in subsequent versions of Java.

Syntactical In Python 2 `print` was a statement. In Python 3 it is now a function, and using it as a statement (without parentheses) will produce an error. Concretely, the following line is valid in Python 2 but invalid in Python 3:

```
print "Hello World"
```

3.2.2 Type-checker level

Source incompatibilities can occur when language designers update parts of the language or libraries that rely or work with specific type information. For example, changing the signatures of components such as interfaces and classes from the built-in language API results in type errors.

Type system updates Java 7 fixed a bug in the method type-inference mechanism [91]. The following code was valid in Java 6 but is now invalid in Java 7:

```
class Test {
    <Z> List<Z> m(List<? super Z> ls) { return null; }

    void test(List<String> ls) {
        List<Integer> i = m(ls);
   }
}
```

The type-inference mechanism previously inferred `Integer` for the type variable `Z` which was problematic since the method `m` receives a list of `String`.

As another example, Java 7 fixed an implementation problem with overloaded methods. Java 6 allowed the following code:

```
class Test {
    int m(List<String> ls) { return 0; }
    long m(List<Integer> ls) { return 1; }
}
```

However, this is problematic due to how generics are implemented by the Java compiler. Generics information are simply ignored ("erased") when compiled to JVM bytecode. As a result, both methods compile to JVM bytecode that will have the same parameter arguments which is disallowed. Java 7 now rejects this code.

Different API contract Java 7 added a method to the interface `javax.lang.model.type.TypeVisitor` to reflect other additions to the language. As a consequence, any user code directly implementing this interface prior to Java 7 fails to re-compile in Java 7. This general issue of improving an API is problematic for APIs designers because it breaks user code. To evolve APIs in a compatible way, Java 8 introduced a new feature called default methods. Default methods bundle implementation code inside a method of an interface and let implementing classes inherit from the code.

However, breaking an API contract can involve more than the parameter and return type of a method. For example, Java 7 changed three methods from the `Path2D` class from non-final to `final`. As a result of this change on the modifiers of these methods, any subclass overriding these methods from the `Path2D` class, which compiled in Java 6 fails to re-compile in Java 7.

3.2.3 Data-flow level

Source incompatibilities can also occur at the data-flow level. For example, a change to the data-flow analysis rules of a language may render user code unreachable which causes compilation to fail in Java.

For instance, we already mentioned in Section 2.3.1 that Java 7 introduced improved checking for rethrown exceptions. Previously, a rethrown exception was treated as throwing the type of the `catch` parameter. Now, when a `catch` parameter is declared `final`, the type is known to be only the exception types that were thrown in the `try` block and are a subtype of the `catch` parameter type.

This new feature introduced two source incompatibilities with respect to Java 6. The code below illustrates the changes. Previously, the statement `throw exception` would throw a `Foo` exception. Now, it throws a `DaughterOfFoo` exception. As a consequence, the catch block `catch(SonOfFoo anotherException)` is not reachable anymore as the `try` only throws a `DaughterOfFoo` exception. Java forbids unreachable code and as a consequence the program would result in a compile error.

```
class Foo extends Exception {}
class SonOfFoo extends Foo {}
class DaughterOfFoo extends Foo {}

class Test {
    void test() {
        try {
            throw new DaughterOfFoo();
        } catch (final Foo exception) {
            try {
                throw exception; // first incompatibility
            } catch (SonOfFoo anotherException) {
                // second incompatibility: is this block reachable?
            }
        }
    }
}
```

3.2.4 Discussion

We discussed that a source incompatible change in its simplest form means that a program which previously compiled, stops compiling after a change is introduced. We argued that source incompatibilities can surface in three main categories which reflect compiler phases: lexer/parser, type-checker and data flow levels. However, compilers are becoming more sophisticated with many different compiler phases. For example, the Scala compiler enumerates over 30 different compiler phases including tail-call elimination and inlining. It is theoretically possible that bugs slip through in the compiler implementation at various different phases, which when fixed, may cause previous programs to stop compiling.

3.3 Data compatibility

We present another aspect of compatibility which we refer to as *data compatibility.* A programming language can have various mechanisms to work with data. These mechanisms may also change during the evolution of the programming language. We classify these mechanisms into two main categories: *serialisation* and *representation.* For example, many languages support a serialisation mechanism to store and transmit data structures. Similarly, there are several different ways to represent fundamental data such as numbers and strings whose representation can also evolve.

3.3.1 Serialization

Serialisation compatibility is concerned with when an instance of a class is serialised, stored and the structure of the class is later changed. The data stored may not be compatible with the new version of the class. Similarly, if an instance of the new class is serialised and then stored, it may not be compatible with the older version of the class.

Java provides a guide outlining compatible and incompatible changes with regard to serialization [89]. For example, deleting a field from a class is an incompatible change because when the instance of the new class is serialised, the data will not contain the value of the field. If the data is read by an older version of the class its field will be set to a default value which may results in behavioural incompatibilities.

3.3.2 Representation

String representation String datatypes in many programming languages were in the past represented as one byte per character based on the ASCII encoding scheme. However, this scheme is limited because several alphabets such as Chinese and Japanese have more than 256 characters. As a result, various other encodings exist to address this representation problem. Unicode is the preferred standard today. Two popular encodings of unicode are UTF-16 which is a 16-bit encoding and UTF-8 which is a 8-bit variable-width encoding that maximizes compatibility with ASCII.

However, the adoption of a different encoding can cause incompatibilities in programming languages. For example, in Python 2 both text and binary data strings were implemented as byte strings. In Python 3, it was decided that all text strings should follow the unicode standard [130]. Mixing text strings with binary data strings (e.g. concatenating) in Python 2 was accepted because they were both represented as 8-bit strings. However, this results in errors in Python 3 because text strings are now unicode, which cannot be concatenated with binary data strings.

Number representation As with strings, there are several ways to represent floating-point numbers. For example, before the IEEE 754 standard was introduced computer manufacturers had their own incompatible formats [64].

3.4 Performance-model compatibility

Some changes to a language may introduce differences with respect to the performance-model assumptions of users both at the speed and memory level. This can be problematic when users for example rely on the fact that an operation or a code pattern is cheap and this is changed in later version of the language. We refer to such changes as *performance-model compatibility*. We argue that users are often unaware of such changes because they typically have no visible output. To help understanding for programmers, we classify such changes into two main categories: *speed*, which is concerned with the execution time of the program and *memory*, which is concerned with the memory consumption of the program.

3.4.1 Speed

Prior to Java 1.7u6 strings were implemented with an internal array `char[]`, an `offset` and a `count` field. Most strings usually start at `offset` zero and `count` is the length of the internal array. However, calling the method `String.substring` created a new string which shared the internal `char[]` with a different `offset` and `count`. This implementation allowed to execute `String.substring` in constant time and save memory by sharing the internal array.

However, this approach caused memory leaks when the users wanted to extract a small string from a very large string. Even if the large string could be discarded, there was still a reference to it from the smaller string via the internal array they both share. Since Java 1.7u6 the internal `offset` and `count` field were removed and `String.substring` returns a copy of the required section from the larger string [92, 95, 94]. As a result, the large string can be discarded and the running application avoids memory leaks. In addition, a string has now also a smaller memory footprint. However, this new implementation implies that the complexity of `String.substring` is now linear (bounded by the size of the section to copy) instead of constant.

3.4.2 Memory

In Java, entries of a `Map` are typically stored in buckets accessed by the generated hashcode of the key. But if many keys return the same hashcode, then performance of search operations will deteriorate, because buckets are implemented as lists with $O(n)$ retrieval. In Java 8 (Oracle's JVM implementation), after a certain threshold is reached, the buckets are now dynamically replaced by sorted binary trees, which have $O(log(n))$ retrieval. However, the trade-off is that sorted binary trees have an increased memory footprint compared to lists [93].

3.4.3 Discussion

We argue that performance-model compatibility is often overlooked by library designers. For example, the Java API does not enforce any guarantees on the performance-model as part of the specification or documentation. By contrast, the Standard Template Library (STL) in C++ supports the idea of *complexity guarantees* as part of the interface of a component. For example, a `vector` is guaranteed to have run-time complexity of element access in constant time. Nonetheless, the specification is only a simplification to make it accessible for users. For example, it does not support memory usage constraints.

Another element of performance compatibility which is increasingly important is *power consumption*. For example, on mobile devices energy consumption affects battery life. Energy consumption in large data centres can significantly limit expansion because of cooling requirements. A survey indicates that programmers lack knowledge and awareness about software energy-related issues. More than 80% of the programmers do not take energy consumption into account when developing software [100]. Increase of IO operations and inefficient data structures are contributors to worse energy consumption.

3.5 Binary compatibility

When a change is introduced to a library or certain other units of a program (e.g. classes, packages or modules), other parts of the program that depend on these units typically also need to be re-compiled to work as intended. Changes that needs re-compilation of dependent parts are called binary incompatible changes [53]. Different languages have different notions of what binary compatibility means exactly.

For example, in contrast to Java, C++ is compiled ahead of time. C++ compilers usually compile class-field accesses into hard-coded offsets based on the layout information of other classes at compile time. This can results in compatibility problems such as the fragile base class problem [86]. In fact, changing the order of the declaration of virtual functions in a class is often binary incompatible because the order of entries in the virtual table of that class will change too. Other units which are using that class will use incorrect offsets. Similarly adding a new public member can change the offsets of previous members and result in a binary incompatibility.

In contrast, the JVM uses references to look up fields and methods at run-time instead of hard-coded offsets which prevents most of these problems.

Nonetheless, the Java language specification defines binary compatibility by “a change to a type is binary compatible with (equivalently, does not break binary compatibility with) pre-existing binaries if pre-existing binaries that previously linked without error will continue to link without error.”

In Java, the runtime linking process consists of three steps: verification, preparation and resolution. It is possible for changes to be binary incompatible at any of these steps. Examples of binary incompatibility changes are well-documented [137]. They include removing a field from a class or changing the signature of a method.

Although undesirable there have been binary incompatible changes during Java’s history. For example, as we mentioned earlier, Java 7 changed three methods from the `Path2D` class from non-final to `final`. Because of this, any subclasses that override these methods would fail to link. In addition, it is also a source incompatible change because re-compiling these subclasses would also fail.

Note that these issues are not only restricted to Java. Other languages which compile to Java bytecode can also run into binary compatibility problems. For example, adding a new method to a Scala trait is binary incompatible. To discover binary compatibility issues, Scala provides a tool called the Scala migration manager that reports binary incompatibilities between two releases of the same library [26].

We highlight five main concerns relating to binary compatibility problems:

1. Related to the layout of data types (e.g. removing members)

2. Related to hierarchy (e.g. removing a supertype)

3. Related to modifiers (e.g. making a class final)

4. Related to API signatures (e.g. changing result or argument type)
5. Related to compiler artefacts (e.g. synthetic methods – methods automatically generated by the compiler rather than being written by hand by the programmer)

We now examine each of these concerns in turn.

3.5.1 Layout

Changes introduced to the layout of a data type can result in binary incompatibilities. For example, re-ordering virtual functions in a C++ class will change the virtual table's offsets. However, other classes are still dependent on the outdated offsets unless they are recompiled. Similarly, adding and removing public fields can cause layout issues.

3.5.2 Hierarchy

Changes introduced to the type hierarchy in a program can result in binary incompatibilities. For example, shrinking a class' (e.g. `A`) set of superclasses and superinterfaces (either directly or transitively inherited) breaks compatibility. In fact, another class `B` may be relying on a subtyping relationship of `A` that is no longer valid after the change.

3.5.3 Modifiers

Changes introduced to the modifiers of program elements can result in binary incompatibilities because the modifiers introduce different contracts. For example, in Java changing a non-abstract class to `abstract` is binary incompatible because attempts to create new instances of that class will produce a linking error (abstract classes cannot be instantiated). Changing a non-final class to be `final` will throw a `VerifyError` as final classes cannot be subclassed.

3.5.4 API contracts

Changes introduced to the contract of an API can result in binary incompatibilities. For example, in Java, changing the result type of a method may result in a `NoSuchMethodError` because the call-site refers to a method that no longer exists. In other words, changing the signature (including parameters) of a method can be seen as deleting the old method and adding a new method with the new signature.

3.5.5 Compiler artefacts

There are other changes that may be considered binary incompatible depending on the compilation process. In fact, some compilers generate metadata or artefacts which are invisible to the programmer but can result in binary incompatibilities. For example, the Java compiler inlines the value of constant fields. If the value of a constant is later changed then classes referencing it will hold the outdated value unless they are re-compiled. As another example, the Java compiler automatically generates *synthetic methods* that are not hand written by programmers. For instance, the Java compiler generates synthetic methods in the context of covariant overriding (i.e. the ability in a subtype to override a method in a supertype and return a more specific type). This mechanism may result in subtle incompatibilities in the context of separate compilation. To illustrate this, we first explain how the Java compiler implements covariant overriding. Next, we show how the problem can arise in the context of separate compilation. Given the following code:

```
class A {
    Object get() { ... }
}

class B extends A {
   @Override
   String get() { ... }
}
```

The JVM does not consider that `B` overrides the method `get` in `A` because they have different signatures based on the return type (`String` and `Object` respectively). As a result, the Java compiler generates an extra overloaded version of the `get` method in the class `B` (called a bridge method) as follows to provide the illusion that `B.get` overrides `A.get`:

```
Object get() {
    return ((B) this).get();
}
```

Now let us assume we have the following two classes:

```
class C {
}

class D extends C {
   String get() { ... }
}
```

and later the class `C` is changed as follows without recompiling `D`:

```
class C {
   Object get() { ... }
}
```

In this scenario, `D.get` looks like it is covariantly overriding `C.get`. However, because the class `D` was separately compiled and the bridge method in `D` not generated, if someone invokes `get()Object` on a `D` object, it will get the implementation from `C` instead of `D`. While this is not strictly a binary incompatibility in the sense that it does not throw a link error, it produces a strange behaviour which requires re-compilation to fix.

3.5.6 Discussion

We showed that there are various sources of backward compatibility including layout, hierarchy, modifiers, API contracts and compiler artefacts. However, these sources can also be classified as “visible” and “invisible”. We highlighted this distinction already when discussing behavioural incompatibilities in Section 3.1.3. For example, compiler artefacts are invisible elements to the programmer as they are not present in the source code. As a result, understanding the compatibility impact of changes is even more difficult for users. The idea of analysing the impact of a requirement change has already been advocated within the software engineering community. In fact, because software are increasingly large and complex, several tools have been developed to support developers in assessing the cost of a requirement change in a software and to identify the related affected artefacts [112, 40, 109]. In this dissertation, we argue that automated tools are also needed for developers to support the comprehension and evaluation of changes introduced by programming language evolution.

3.6 Security compatibility

Some changes can affect the security of running applications. We categorise these changes under *security compatibility.* We distinguish two categories of security related changes:

1. changes that downgrade the security of the application

2. changes that upgrade the security of the application, but may stop the application running as before by introducing additional restrictions

For example, PHP 5.3 deprecated the the *magic quotes* functionality which was later completely removed in PHP 5.4. Magic quotes was a functionality to reduce injection exploits by automatically escaping special characters with a backslash in incoming data to the PHP script. Unfortunately magic quotes had several problems which gave users a false sense of security. For example, they did not fully protect against cross-site scripting attacks. However, its removal may have opened new security bugs in existing code that was previously protected against. For instance, beginner programmers frequently relied on it to avoid basic SQL injections. Another example of downgrading security by introducing a change occurred in the OpenSSL package on Debian. A line of code was commented out in the implementation to remove an uninitialised data warning. However, this resulted in making the random number generator more guessable and the implementation less secure [25].

In general, security updates are intended to upgrade the security of a system. However, it can also result in incompatibilities. For example, recently, Java 8 mandated that certificates are blocked if they contain RSA keys of less than 1024 bits in length. The compatibility states that “With this key-size restriction, those who use X.509 certificates based on RSA keys less than 1024 bits will encounter compatibility issues with certification path building and validation. This key size restriction also impacts JDK components that validate X.509 certificates, for example signed JAR verification, SSL/TLS transport, and HTTPS connections.” [96]

3.7 Summary

In this chapter, we have argued that backward compatibility is more subtle than it appears and is an important topic that concerns developers. Our claim is confirmed by a recent study which shows that programmers lack awareness of compatibilities beyond compile errors [52].

We developed a classification of backward compatibility across six categories by surveying bug reports and compatibility guide from several programming languages including Java, Python and PHP. Our classification builds upon previous work [44, 48] which introduced *behavioural compatibility*, *source compatibility* and *binary compatibility*.

Concretely, we made two contributions. First, we established that changes can also be categorised by *performance-model compatibility*, *data compatibility* and *security compatibility*. Second, we presented a finer-grained detail of the different kinds of behavioural compatibility, source compatibility and binary compatibility. One aim of this work is to contribute a checklist for programmers to better understand the possible compatibility issues when migrating their codebases to different language and library versions.

Furthermore, we discussed that the idea of analysing the impact of a new requirement change has already been advocated by the software engineering community. In fact, because software is increasingly large and complex, several tools have been developed to support developers in assessing the cost of a requirement change in a software and to identify the related affected artefacts [112, 40, 109]. In this dissertation, we argue that automated tools are also needed for developers to support the comprehension and evaluation of changes in the context of programming language evolution. With this in mind, subsequent chapters focus on developing program comprehension tools to support programming language evolution. Particularly, we look at source and behavioural changes.

Chapter 4

Source code queries with graph databases

In the previous chapters, we argued that programming language evolution is an important topic. In fact, it affects developers because they have to maintain and migrate code written in older language versions. We discussed that often changes between language versions introduce subtle incompatibilities that are hard to understand yet manually locate in source code. Consequently, programming language evolution can be time-consuming and expensive for programmers to deal with in practice.

In this chapter, we present an automated tool to help language designers and developers gather information about source code which would be difficult to collect manually. We show how this tool is useful for program comprehension and in the context of programming language evolution for language designers. Next chapters focus on more subtle issues that arise with locating behavioural incompatibilities.

4.1 Introduction

Large programs are difficult to maintain. Different modules are written by different software engineers with different programming styles. However, enhancements to a system often require modifying several parts at the same time. As a consequence, software engineers spend a large amount of their time understanding the system they are working on.

Various automated tools have been developed to assist programmers to analyse their programs. For example, several code browsers have been developed to help program comprehension through hyperlinked code and simple queries such as viewing the type hierarchy of a class, or finding the declaration of a method. These facilities are nowadays available in mainstreams IDEs but are limited to fixed query forms and lack flexibility.

Recently, several source code querying tools have been developed to provide flexible analysis of source code [45, 50, 2, 11, 70, 66]. They let programmers compose queries written in a domain specific language to locate code of interest. For example, they can be used to enforce coding standards (e.g. do not have empty blocks) [13], locate potential bugs (e.g. a method returning a reference type `Boolean` and that explicitly returns `null`) [8], code to refactor (e.g. use of deprecated methods), or simply to explore the codebase (e.g. method call hierarchy).

Such tools are also useful for programming language designers to perform program analysis on corpora of software to learn whether a feature is prevalent enough to influence the evolution of the language. For example, a recent empirical study made use of a source code query language to investigate the use of overloading in Java programs [62].

Typically, source code querying tools make use of a relational or deductive database to store information about the program because it is inconvenient to store information about large programs in main memory. However, they typically let users query only a subset of the source code information (e.g. class and method declarations but not method bodies) because storing and exposing further information affects the scalability of the querying system. As a result, existing querying facilities have different trade-offs. We distinguish between two categories. On one hand, some systems scale to programs with million lines of code but provide limited information about the source code. For example, statement level information is discarded. On the other hand, some systems store more information about the source code but do not scale very well to large programs.

In this chapter, we argue that it is possible to build a system which allows expressive queries on the full source code of a program and that also scales to large programs.

We observe that programmers are familiar with compiler-like graph structures such as the parse tree, control flow graph and type hierarchy. We therefore suggest that the *graph data model* [33] is a natural conceptual fit for representing source code information. We present a model for Java source code that can be queried for expressive queries involving full structural, type and flow information. For example, we can query statement level information, the actual type of any expression, the type hierarchy, the call graph of any method, read/write dependencies of variables and fields as well as liveness information. In addition, we explain how our model can be easily extended via overlays and queried using graph traversal operations.

We implemented a source code querying system based on this model using a *graph database* (i.e a database specialised for storing and traversing graph-like structures). Experiments show that our prototype based on Neo4j [18] scales to programs with million lines of code and compares well to existing work.

The main contributions of this chapter are as follows:

- The identification of different views of a program that are useful to query for common software engineering tasks and language design questions (Section 4.3). We make these available as graph *overlays* (Section 4.4).

- A model for representing Java source code information based on the graph data model that can support structural, type and flow queries (Section 4.4 and 4.5).

- An implementation of a source code query system based on this model using a graph database (Section 4.6).

- A detailed set of experiments that demonstrates that our prototype is expressive and scales with increased program sizes (Section 4.7).

4.2 Background

In this section, we describe the *graph data model* and *graph databases*.

4.2.1 Graph Data Model

The graph data model encodes entities and relationships amongst them using a *directed graph structure* [33]. It consists of sets of nodes and edges, where nodes represent entities and the edges represent binary[1] relationships between these entities. Nodes and edges can be labelled, typically with the name of the entity they represent, but possibly with a tuple of name and further values, such as "weight:3". For our purpose, we will refer to such tuples as *properties* on nodes or edges.

As an example, a node may have a property `nodeType` holding the string/enumeration value `JCBinary` to represent a binary expression. This node would typically be connected to two other nodes via two distinct edges labelled `LEFT` and `RIGHT` to connect the left and right child nodes of a binary expression.

By contrast, the *relational model* represents entities as tuples that are grouped into formal relations. Relationships between entities are encoded by combining relations based on a common attribute.

4.2.2 Graph Databases

Databases supporting the graph data model are often referred to as *graph databases.* They can be accessed via queries expressing graph operations such as traversal, pattern matching and graph metrics. In Section 4.5 we discuss a graph query language called *CYPHER*, which is used by the Neoj4 graph database.

By contrast databases supporting the relational model are referred to as *relational databases.* They are typically accessed using SQL as the query language.

There are two fundamental differences between relational and graph databases which we exploit in our work.

First, graph databases provide index-free adjacency. In other words, records themselves contain direct pointers to list of connected nodes. This obviates the need to combine relations based on a common attribute to retrieve connected records. Vicknair et al. evaluated the performance of a graph database against a relational database. They report that graph databases executed better for traversing connected data, sometimes by a factor of ten [134].

Second, relational databases depend on a schema to structure the data whereas graph databases typically do not have this requirement (i.e. they support *semi-structured data*). In practice, it means that in a graph database all records need to be examined to determine their structures, while this is known ahead of time in a relational database. This potential source of inefficiency is addressed in some graph databases which provide index facilities to map certain structures known ahead of time to a list of records. In addition, because graph databases do not depend on a schema, they can be more flexible when the structure of records needs to be changed. In fact, additional properties can be added to or removed from records seamlessly, which facilitates the addition of user-defined overlays. In relational databases, a modification of the schema is required before altering the structure of records.

[1] In principle the formulation allows hyper-graphs with hyperedges of having more than two nodes, but we only find the need to use ordinary directed edges.

4.3 Program Views

In this section, we identify various syntactic and semantic properties of programs which are useful in searches. We make these available as *overlays* in the graph model (see Section 4.5) so we can form queries combining various concepts, such as "find a local variable which is declared in a method called recursively and which has multiple live ranges". These overlay properties can be seen as database *views*. For each property, we discuss their use for common software engineering tasks and language design questions.

- **Text/lexical**
  These queries express grep-like queries to match pattern on the source code. Such queries have an expressive power limited to regular expressions. Such queries are useful to check certain coding standards (e.g. check whether variable declarations follow a naming convention) and also for *program comprehension* (e.g. track certain keywords in the code base). There exist many regular-expression-based code search facilities such as Google code search [10].

- **Structure**
  Structural queries operate on the abstract syntax tree (AST). These queries can be classified in two categories *global structure* (e.g. querying for abstract method declarations within a specific class) and *statement-level* (e.g. querying for a chain of `if` statements to refactor into a `switch`) [103].

- **Semantic Hierarchy**
  These queries explore semantic hierarchies among program entities. For example, a query language could provide support for the overriding or implementation hierarchy of methods. In addition, it could provide support for the type hierarchy defined in the code base. This is useful both for *program comprehension* (e.g. to find all concrete classes of an interface) and *optimisation* (e.g. perform class hierarchy analysis and eliminate dynamic dispatches).

- **Type**
  Here we effectively query a type-decorated AST of the program. For example, in the context of *program comprehension* one could query method calls on a variable that has a particular type. Such a query requires the availability of variables declaration's type. In addition, some *coding standards* require type information (e.g. avoid the use of covariant arrays).

- **Control Flow**
  Control flow queries investigate the possible execution paths of a program. Such information can be useful in the context of *program comprehension*. For instance, to get a high-level overview of an algorithm or to compute software metrics such as the cyclomatic complexity. In addition, one may which to explore a codebase by querying for all method calls made inside a program entry point (e.g. `main()`) or to analyse which parts of a module are calling a methods from a database API. Such query requires access to the method call graph of a program.

- **Data Flow**
  Data flow queries examine the flow of values in a program. For example, one could query whether a variable is used in the future by inspecting the liveness set of

each statement. This is useful for *optimisation* and for removing unused variables. In addition, in the context of *program comprehension* one could query whether a method reads or write a specific field, the reaching definition of an assignment, or the number of items a variable may point to.

- **Threading primitives**
  Many programming languages including Java provide threading primitives at the language level (e.g. `synchronized` and `wait`). Such primitives could be specified as an overlay to query concurrency properties of a program such as what parts of the source code spawns and joins threads.

- **Compiler Diagnostics**
  It is sometimes useful to access extra information generated by the compiler. For example, one may want to query the list of warnings associated with a specific block of code. In addition, in the context of *program comprehension* it could be useful to query various decisions made by two versions of a compiler to identify incompatibilities. For example, Java 8 brings more aggressive type inference compared to Java 7. There has been discussion about how to identify possible incompatibilities by inspecting the inference resolutions log generated by the Java compiler (the default `javac`) [12].

- **Runtime Information**
  Run-time information is useful in order to investigate the impact of code at run time such as its memory consumption and CPU usage. For example, synchronized blocks are often responsible for performance bottleneck. In fact, there are query technologies that can analyse a Java heap to facilitate application troubleshooting [19].

- **Version History**
  Certain type of queries can only be answered by analysing the version history of a codebase. For example, in the context of *program comprehension* it could be useful to query for methods that have been updated the most, to find all modifications made by a specific commiter or simply to perform code diffs [73]. Such information is also useful for *programming language research* to investigate the adoption of certain development practices, new libraries or language features over time [102]. In addition, recent work suggested the use of version history to mine aspects from source code [43].

4.4 A Graph model for source code

Our model to represent source code information consists of several compiler-like graph structures. We enhance these structures with additional relations, which we call *overlays*, to enable and facilitate queries over multiple program views.

In our model, source code elements are represented as nodes and the relationships amongst them as edges. We focus on Java. However, our methodology can be applied to any object-oriented language. The design of our model was guided by our motivation to support full structural information about the source code as well as type and flow information.

We provide some examples on how to query our model. Data manipulation is explained in more detail in Section 4.5.

4.4.1 Abstract Syntax Tree

The AST of a program is a directed acyclic graph where the nodes represent source code entities and the edges represent directed parent to child relationships. The AST fits therefore naturally with the graph data model.

The Java compiler defines different types of AST node types to represent different source code elements. For example, a method declaration is represented by a class named `JCMethodDecl`, a wildcard by a class named `JCWildcard`. We map the AST nodes into nodes in our graph model and we use these class names to identify the nodes. We store the names in a property called `nodeType` (an enumeration containing `JCMethodDecl` etc) attached to each node.

Some AST node have fields pointing to other AST nodes to represent parent-child relationships. For example, a binary expression is represented by a `JCBinary` class, which contains a field named `lhs` pointing to its left child expression and another field named `rhs` pointing to its right child expression. We map these names to labelled edges. As an example, in our model `JCBinary` is mapped to a node with a property (`nodetype:"JCBinary"`), two edges labelled respectively `LHS` and `RHS` connecting two separate nodes with a property (`nodetype:"JCExpression"`).

The complete list of relationships consists of around 120 different labelled edges and can be accessed on the project home page [29]. It includes `IF_THEN` to represent the relationship of an if statement and its first branch, `IF_ELSE` to represent the relationship of an if statement with its last branch, `STATEMENTS` to represent a block enclosing a list of statements etc. Note that we also add a relation `ENCLOSES` on top of these (i.e. an overlay) to represent the general relationship of a node enclosing another one.

This model lets us retrieve source code elements using the `nodeType` property as a search key. Searching for certain source code patterns becomes a graph traversal problem with the correct sequence of labelled edges and nodes. For example, finding a while loop within a specific method can be seen as finding a node `JCMethodDecl` connected to a `JCWhileLoop` with a directed labelled edge `ENCLOSES`.

Additionally, we define two properties on the nodes that are useful for software engineering metrics queries: `size` (e.g. number of lines of code or character count that this node represent in the source file) and `position` (e.g. starting line number and/or offset of this node in the source file). These are implemented easily as the javac front-end provides them for every syntactic object.

4.4.2 Overlays

The most obvious graph model of source code is the abstract syntax tree. However, even finding the source-language type of a node representing a variable use is difficult as we need an iterative search outwards through scopes to find its *binding* occurrence in a definition which is where the type is specified.

What we need is an additional edge between a variable use and its binding occurrence—this additional relation is an *overlay*. For example, we connect method and constructor invocations to their declarations with a labelled edge `HAS_DECLARATION` and variables and field uses to their respective definitions with a labelled edge `HAS_DEFINITION` to access these directly. Similarly we might want to precompute a program's call graph and store it as an overlay relation instead of searching for call nodes on demand.

Sometimes it is also desirable to distinguish different instances of a same labelled edge. For example, while a class declaration contains a list of definitions, it is useful to distinguish between field and method definitions to facilitate querying. This is why we distinguish these with `DECLARES_FIELD` and `DECLARES_METHOD` overlays.

Similarly, it is difficult to know ahead of time all possible information that users may want to query. For example, users may want to query for overloaded methods. Such a query could be defined by processing all methods nodes available in our model and grouping them by fully-qualified name and returning the groups that occur more than once. However, this query would be a lot more efficient if methods nodes had a pre-computed property stating that they are overloaded methods.

As explained earlier, the graph data model relies on semi-structured data so new overlays can be added without the schema to change. As a result, our model can easily cache results of queries as additional relations or nodes.

We now describe the pre-computed overlays in our model corresponding to the program views described in Section 4.3.

Type Hierarchy

We overlay the type hierarchy defined in the program on top of the AST. We connect nodes that represent a type declaration with a directed edge to show a type relation. These edges are labelled to distinguish between an `IS_SUBTYPE_IMPLEMENTS` or `IS_SUBTYPE_EXTENDS` relation. These overlays lets us map the problem of finding the subtypes of a given type as a graph path finding query. As a result, we do not need to examine class and interface declarations to extract the information. Note that we prefer to add overlays for 1-step relations (e.g. direct subtype) as transitive closure can be simply expressed and computed by a graph query language (see Section 4.5).

Override Hierarchy

Similarly, we overlay the override hierarchy of methods defined in the program on top of the AST. We connect method declaration nodes in the AST with the parent method they override using labelled edges `OVERRIDES`. As a result, we can query the override hierarchy directly instead of examining each class and method declaration to extract the information.

Type Attribution

We overlay a property called `actualType` on each expression node in the AST. It holds the string representation of the resolved type of an expression. This facilitates common source code queries such as finding a method call on a receiver of a specific type. Indeed, this query would simply retrieve the method invocation nodes (labelled `JCMethodInvocation`) and check that the property `actualType` is equal to the desired type. Similarly, one can find covariant array assignments by comparing the `actualType` property of the left-hand and right-hand side of an assignment node (labelled `JCAssign`).

The Java compiler also defines an enumeration of different type kinds that an expression can have. These includes `ARRAY`, `DECLARED` (a class or interface type), `INT`, `LONG`. We store these values in a `typeKind` property.

Method Call Graph

We construct the method call graph by extracting the method invocations from the body of each method declaration. Each method invocation is resolved and associated with its method declaration based on two filters: the static type of the receiver (conservative call graph) and all possible subtypes of the receiver static type (rich call graph). We connect a method invocation node with its resolved method declaration(s) with a labelled edge `CALLS` for the conservative call graph and `CALLS_DYN` for the rich call graph.

Typical source code queries such as finding all the methods called within the entry point of a class can be seen as a traversal query on the generated call graph. In addition, we can easily find recursive methods by finding method declarations nodes with a `CALLS` loop.

Data Flow

Dataflow analysis is a vast topic; this work describes only a simpler set of data flow queries. We connect each expression with the declaration of the variables that they use using a `GEN` edge. Similarly, variables that are necessarily written to are connected to the expression with a `KILLS` edge. Such information is useful for performing certain data flow analysis such as live variable analysis. For writes, we consider the conservative set of variables that must be written to.

Next, we construct the read and write dependency graphs for each block. We inspect block declarations (e.g method declarations, loops) to extract any field and variable accesses. We connect the blocks to the field or variable declarations that they read from or write to using `READS` and `WRITES` edges. The `READS` set of a block can be seen as the union of all the outgoing nodes connected with a `GEN` edge from all the expressions contained inside that block. Similarly, the `WRITES` set can be seen as the union of all outgoing nodes connected with a `KILLS` edge from all the expressions contained inside that block. This additional overlay is useful to easily compose program comprehension queries such as finding out which fields are written to by a given method declaration.

4.5 Source code queries via graph search

In this section, we describe how one can write source code queries using our model. We show that a query language specialised for graph traversal that supports *graph pattern matching*, *path finding*, *filtering* and *aggregate functions* can express all queries expressible in existing Datalog approaches.

Hajiyev et al. highlighted that a source code query language needs to support recursive queries to achieve high expressiveness [66]. Indeed, certain queries such as traversing the subtype hierarchy or walking tree structures require recursion. To this end, they suggested Datalog, which supports built-in transitive closure. *Graph pattern matching* (i.e the ability to match certain parts of a graph in a variable) can be seen as binding a set of nodes in a variable based on the values of node properties and labeled edges. *Path finding* operations have also built-in transitive closure in order to traverse the full graph.

We show examples of source code queries using *CYPHER*, a graph query language for the Neo4j database [18]. Its use obviates the need for complex translation to other database query languages such as Codequest [66]. CYPHER also supports aggregate functions such as `COUNT()`, `SUM()`, which appear only as extensions to Datalog. Such operations are useful to express software metric queries because they often need to apply arithmetic operations on a group of data.

The next subsections describe in detail several of source code queries written in CYPHER. The examples combine program comprehension queries and research queries of interest to language designers.

4.5.1 Implementing java.lang.Comparator

We introduce the syntax of CYPHER by showing a simple query to retrieve all classes implementing the interface `java.util.Comparator` in Figure 4.1 (Listing 1).

The `START` clause indicates a starting point in the graph. We then iteratively bind in the variable `p` to all the nodes that have a property `nodeType` equal to `ClassType` by looking up the index of nodes called `node_auto_index`. Next, we use the `MATCH` clause to generate a collection of subgraphs matching the given graph pattern. Specifically, we look for subgraphs starting at a node bound to a variable `n` which has an outgoing edge labelled `IS_SUBTYPE_IMPLEMENTS` connected to `p`. Next, we eliminate any subgraphs where `p` is not of type `java.util.Comparator` by using a `WHERE` clause on the property `fullyQualifiedName`. Finally, we return the elements of all the matched subgraph as a list of tuples.

Table 4.2 shows an extract of the output from executing this query on the source code of the program *Hsqldb*.

4.5.2 Extending java.lang.Exception

We can modify this query to find all classes that are directly or indirectly subtypes of `java.lang.Exception` as shown in Figure 4.1 (Listing 2). We generate a collection of all subgraphs starting at a node bound to the variable `n` that is connected to a node `m` that represents `java.lang.Exception`. We use the asterisk (`*`) to specify a form of transitive closure: here that the path to reach `m` can be of arbitrary length as long as it is solely composed of edges of type `IS_SUBTYPE_EXTENDS`. Finally, we bind all the subgraphs that were matched into a variable `path`, which we return.

Table 4.3 shows an extract of the output from executing this query on Hsqldb. The class `FileCanonicalizationException` extends `BaseException`, which itself extends `java.lang.Exception`

4.5.3 Recursive methods with their parent classes

In Figure 4.1 (Listing 3) we show how to search for classes containing recursive methods. First, we look up the index of nodes to find all method nodes that have the property `nodeType` equal to `JCMethodDecl` and iteratively bind them to the variable `m`. Next we specify a graph pattern using the `MATCH` clause that generates a collection of all subgraphs starting at a node `c` (a class declaration) that is connected to `m`. In addition, we specify that the method `m` is calling itself by specifying an edge `CALLS` from `m` to `m` (conservative call graph).

Listing 1

```
START p=node:node_auto_index(nodeType='ClassType')
MATCH n-[:IS_SUBTYPE_IMPLEMENTS]->p
WHERE p.fullyQualifiedName='java.util.Comparator'
RETURN n,p;
```

Listing 2

```
START m=node:node_auto_index(nodeType='ClassType')
MATCH path=n-[r:IS_SUBTYPE_EXTENDS*]->m
WHERE m.fullyQualifiedName='java.lang.Exception'
RETURN path;
```

Listing 3

```
START m=node:node_auto_index(nodeType='JCMethodDecl')
MATCH c-[:DECLARES_METHOD]->m-[:CALLS]->m
RETURN c,m;
```

Listing 4

```
START n=node:node_auto_index(nodeType ='JCWildcard')
RETURN n.typeBoundKind , count(*)
```

Listing 5

```
START n=node:node_auto_index(nodeType='JCAssign')
MATCH lhs<-[:ASSIGNMENT_LHS]-n-[:ASSIGNMENT_RHS]->rhs
WHERE
    has(lhs.typeKind) AND lhs.typeKind='ARRAY' AND
    has(rhs.typeKind) AND rhs.typeKind='ARRAY' AND
    lhs.actualType <> rhs.actualType
RETURN n;
```

Listing 6

```
START m=node:node_auto_index(nodeType ='JCMethodDecl')
MATCH c-[:DECLARES_METHOD]->m
WHERE m.name <> '<init>'
WITH
    c.fullyQualifiedName+":"+m.name as fullyQualifiedMethod,
    count(*) as overloadedCount
WHERE overloadedCount > 1
RETURN fullyQualifiedMethod, overloadedCount
ORDER BY overloadedCount DESC;
```

Figure 4.1: Examples of source code queries written in CYPHER

4.5.4 Wildcards

Parnin et al. investigated the use of generics by undertaking a corpus analysis of Java code [102]. In this example, we show how a query can investigate the use of generic wildcards. We would like to group the different kinds of wildcards by the number of their occurrences which appear in the source code. CYPHER provides aggregate functions such as `count(*)` to count the number of matches to a grouping key. We can use this function in a `RETURN` clause to count the number of occurrences based on typeBoundKind property of nodes of type `JCWildcard` as shown in Figure 4.1 (Listing 4). Figure 4.4 shows a possible output of running this query.

n.typeBoundKind	count(*)
UNBOUNDED_WILDCARD	67
EXTENDS_WILDCARD	32

Figure 4.4: Result for grouping Java wildcards by kinds

Start node (n)	End node (p)

`{nodeType:"JCClassDecl",`
`lineNumber:36,`
`position:1654,size:473,`
`simpleName:"StringComparator",`
`fullyQualifiedName:"org.hsqldb.lib.StringComparator"}`
	`{nodeType:"ClassType",`
	`fullyQualifiedName:"java.util.Comparator"}`

`{nodeType:"JCClassDecl",`
`lineNumber:56,`
`position:2444,size:9564,`
`simpleName:"SubQuery",`
`fullyQualifiedName:"org.hsqldb.SubQuery"}`
	`{nodeType:"ClassType",`
	`fullyQualifiedName:"java.util.Comparator"}`

`{nodeType:"JCClassDecl",`
`lineNumber:86,`
`position:3576,size:417,`
`simpleName:"",`
`fullyQualifiedName:"<anonymous java.util.Comparator>"}`
	`{nodeType:"ClassType",`
	`fullyQualifiedName:"java.util.Comparator"}`

Figure 4.2: Example output for the query *implementing java.lang.Comparator* on the source code of the program *Hsqldb*

Node (n)	Outgoing relationship (r)

`{nodeType:"JCClassDecl",`
`lineNumber:2002,`
`position:83668,size:1388,`
`simpleName:"FileCanonicalizationException",`
`fullyQualifiedName:"org.hsqldb.persist.LockFile.FileCanonicalizationException"}`
	`:IS_SUBTYPE_EXTENDS`

`{nodeType:"JCClassDecl",`
`lineNumber:1937,`
`position:81667,size:1684,`
`simpleName:"BaseException",`
`fullyQualifiedName:"org.hsqldb.persist.LockFile.BaseException"}`
	`:IS_SUBTYPE_EXTENDS`

`{nodeType:"ClassType",fullyQualifiedName:"java.lang.Exception"}`	

Figure 4.3: Example output for the query *directly and indirectly extending java.lang.Exception* on the source code of the program *Hsqldb*

4.5.5 Covariant array assignments

We recently conducted a corpus analysis to investigate the use of covariant arrays and found that these are rarely used in practice [128]. In Figure 4.1 (Listing 5), we show how one could find uses of covariant arrays in assignment context.

First, we look up nodes of type `JCAssign`, which represent an assignment in the AST. We iteratively bind them to the variable `n`. Next, we use a `MATCH` clause to decompose an assignment into its left and right children, which we bind respectively in the `lhs` and `rhs` variable. We ensure that these nodes are both tagged as holding an expression of type `ARRAY` by restricting the `typeKind` property using a `WHERE` clause. Finally, we also ensure that the resolved type (e.g. `String[]`) of the left hand side and right hand side of the assignment are different by checking the `actualType` property, which hold the string representation of the type of expression.

4.5.6 Overloaded Methods

Recent work has investigating how Java programmers make use of overloading by undertaking a corpus analysis of Java code [62]. This research question can be partially answered by composing a query which retrieves overloaded methods. Figure 4.1 (Listing

6) shows how to generate a report of what method names in which classes are overloaded along with the degree of overloading.

First, we bind all nodes of type `jCMethodDecl` to variable `v`. We then generate all subgraphs connecting `n` with a node `c` (a class declaration) with a `DECLARES_METHOD` relation. We add an additional constraint using a WHERE clause to eliminate subgraphs where the method nodes are constructors (methods named `<init>`). Next, we generate the fully qualified name of the method by concatenating the fully qualified class declaration name with the method name. We use this as the grouping key for the `count(*)` aggregate function. Finally, we return the fully qualified method names that appear more than once (i.e overloaded methods) together with how many times they occur.

4.6 Implementation

In this section, we give an overview of the implementation of our source code querying system.

Our current prototype consists of a Java compiler plugin to analyse source code files and a Neo4j graph database to store the source code information converted in our graph model. We selected Neo4j [18] as our database backend because it is established in the community, has a Java API and also provides a graph query language that supports the necessary features described in Section 4.5. The source code of our prototype (1300 lines of Java code) can be accessed on the project homepage [29].

The first part of our system is responsible for parsing Java source code files, analysing them and converting them into our graph model. To this end, we developed a Java compiler plugin that accesses the Java compiler structures and functionalities via API calls [126]. We used this mechanism because a compiler plugin interacts in harmony with existing building tools such as *ant* and *maven* that rely on the Java compiler. Users of our tool can therefore compile their programs normally without running external tools. However, the Java compiler requires the entire project to compile before it can be processed. There are alternatives such as Eclipse JDT [7] that do not have this requirement but that do not integrate so smothly with standard building tools.

Our compiler plugin traverses the Abstract Syntax Tree (AST) of each source code file. It translates the AST elements and their properties to graph nodes in Neo4j. Links between AST elements are translated to labelled edges between the created nodes. We retain the full AST and do not throw away any information. Additionally, we implemented the type hierarchy, type attribution and call graph overlays described in Section 4.4.

Neo4j provides indexing facilities that map a key to a set of records. We use these facilities to index nodes based on the `nodeType` property in order to improve the speed of specific AST node retrieval queries. Finally, source code queries can be executed by writing CYPHER queries through the shell interface provided with the Neo4j server.

4.7 Experiments

In this section, we evaluate the scalability and performance of our system. We describe a set of experiments and their results demonstrate that our system scales well.

First, we evaluate the performance of queries related to code exploration. Next, we investigate the performance of queries related to programming language evolution that make use of richer source code information. Finally, we also compare performance of certain queries supported by JQuery [70].

4.7.1 Experimental Setup

Our framework intercepts all calls to *javac* by replacing its binary on the standard search path with an executable file that calls *javac* with our compiler plugin. As a result, any Java build tools (such as *ant* or *maven*) that use *javac* to compile files will automatically process the source code in the graph database.

We selected four Java open source programs of various sizes. We summarise them in Table 4.5. In addition, we created a corpus of 12 Java projects totalling 2.04 million lines of code, which can be accessed on the project home page. It includes *Apache Ant*, *Apache Ivy*, *AoI*, *HsqlDB*, *jEdit*, *Jspwiki*, *Junit*, *Lucene*, *POI*, *Voldemort*, *Vuze* and *Weka*.

We selected these programs from the Qualitas Corpus, which is a curated collection of software systems intended to be used for empirical studies of code artefacts [120]. Many studies in Java have used this corpus. We used a working subset of the corpus because several of the projects included in the corpus did not compile or were missing build instructions which is necessary to perform our compiler analysis.

We ran the experiments on two machines with Neo4j standalone server 1.9.M04 and OpenJDK 1.7.0 using 4GB of heap size:

- A typical personal computer used for programming tasks: a MacBook Pro with 2.7GHz Intel Core i7, 8GB of memory and running OSX 10.8.2

- An Amazon EC2 instance (m3.2xlarge) with 30GB of memory and EBS storage running Linux. We provide the virtual image on the project homepage for reproducibility.

We ran the database server with default settings. However, it is possible to tune certain configuration parameters such as memory usage to gain additional performance.

Our experiments consist of running different queries on the programs and report the reponse time. We distinguish between a *cold* system and a *warm* system to investigate the impact of caching on performance.

- We report the response time of a query after restarting the querying system each time (*cold*).

- We run the same queries twice and report the faster response time (*warm*).

In a real scenario, it is possible for a developer to run the same query several times. In practice, a production source-querying system could even warm the cache ahead of time with common queries to enhance performance. Having similar sub-queries in common will also benefit from the cache.

Application	Description	# of Java files	LoC
Junit 4.2	Java unit test framework	79	3206
Apache Ivy 4.2.5	Dependency manager	601	67989
Hsqldb 2.2.8	Relational database engine	520	160250
Vuze	Bitorrent client	3327	509698
Corpus	12 open source software	13559	2038832

Figure 4.5: Summary of Java projects used for the experiments

4.7.2 Queries

Code Exploration Queries

Source-code querying systems are useful to developers in order to rapidly explore a codebase by retrieving some elements. We ran the following queries that explore a given codebase (the syntax for them is described in Section 4.5):

1. Find all classes that directly implement `java.util.Comparator` (Figure 4.1, Listing 1)

2. Find all classes that directly or indirectly extend `java.lang.Exception` (Figure 4.1, Listing 2)

3. Find all recursive methods *together with* their parent classes (Figure 4.1, Listing 3)

Programming Language Evolution Queries

Source code querying systems can also be used by programming language researchers to retrieve usage information about certain features or patterns. For each project, we ran the following research queries as these were investigated in previous work (the syntax for them is described in Section 4.5):

1. Report the ratio of the usage of generic wildcards (Figure 4.1, Listing 4)

2. Find all covariant array assignments (Figure 4.1, Listing 5)

3. Find all overloaded methods (Figure 4.1, Listing 6)

4.7.3 Results

We show the absolute response time to the queries in Figure 4.7. Queries 4 and 6 have a response time less than half a tenth of a second. This shows that while our system can efficiently traverse graph structures, aggregate operations are also efficiently executed. Figure 4.6 presents the execution time expressed as ratios relative to the program *Hsqldb*. The results shows that the query response times scale linearly with increasing program size.

We found that running queries with a hot cache decreases response time by a factor of 3x to 5x compared to a cold cache. Finally, we also found that a larger machine does not directly improve performance of queries. We note a net improvement for the largest programs (Corpus) for query 3 and 5 with a cold cache.

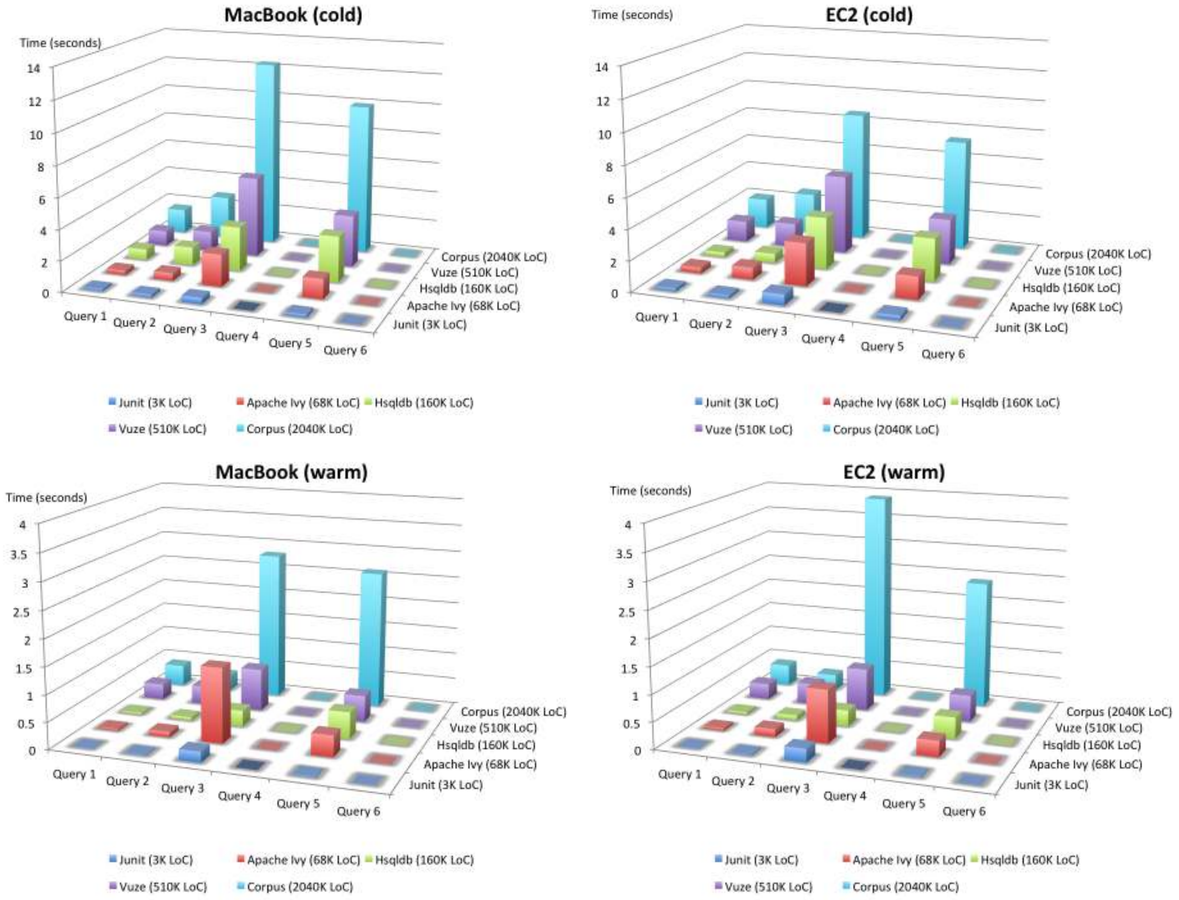


Figure 4.7: Absolute queries times on MacBook and EC2

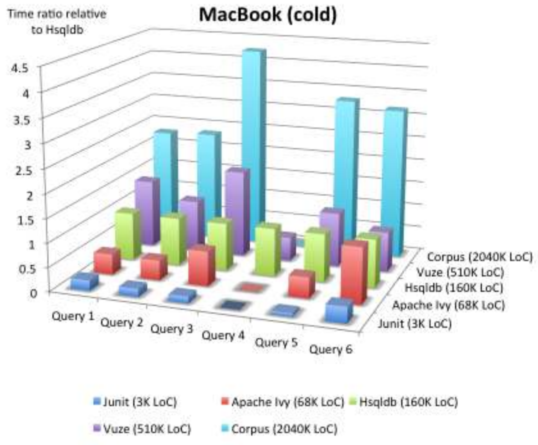


Figure 4.6: Queries times on MacBook relative to Hsqldb (cold)

We report memory usage, number of nodes, properties and relationships about the generated graph as described by the system administration of Neo4j in Table 4.8.

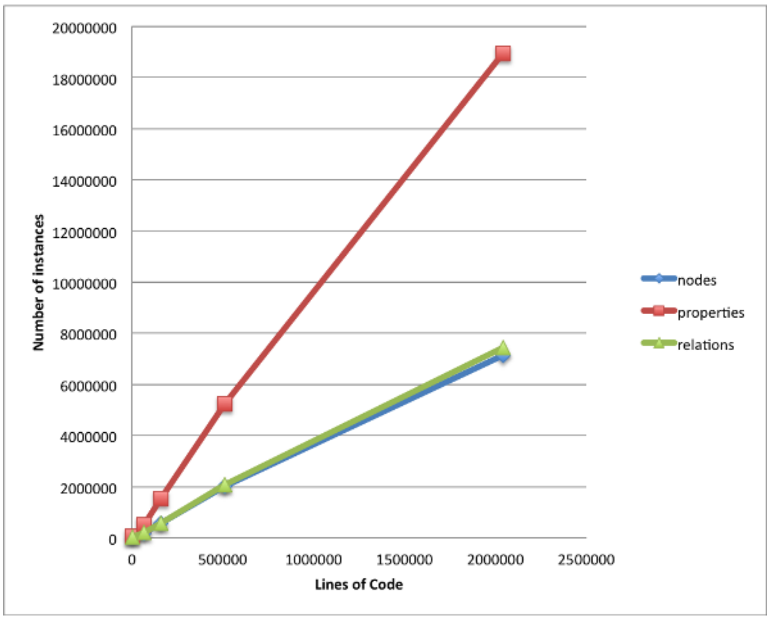


Figure 4.9: Graph metics

Application	Memory Storage (MiB)	# Nodes	# Properties	# Relations
Junit	14.72	16209	43874	16678
Apache Ivy	143.94	188296	514755	196389
Hsqldb	373.56	562193	1535100	583404
Vuze	1410	2002771	5225382	2073828
Corpus	5043.73	7126121	18937054	7424652

Figure 4.8: Metrics for generated graphs of Java projects

Figure 4.9 compares graph metrics and Figure 4.10 plots the memory storage usage. It shows that the memory scales comparatively as the number of lines of code increases. The number of properties in the graph is roughly a constant multiplier over the number of nodes and relations. The objective of our evaluation was to explore whether our source code graph data model implemented using a graph database could act as suitable system for source code querying. We compare to previous work which has conducted benchmarks on different graph databases including Neo4j to evaluate factors such as graph memory consumption and runtime performance for graph algorithms (e.g. traversal and connectedness) [71, 81, 37].

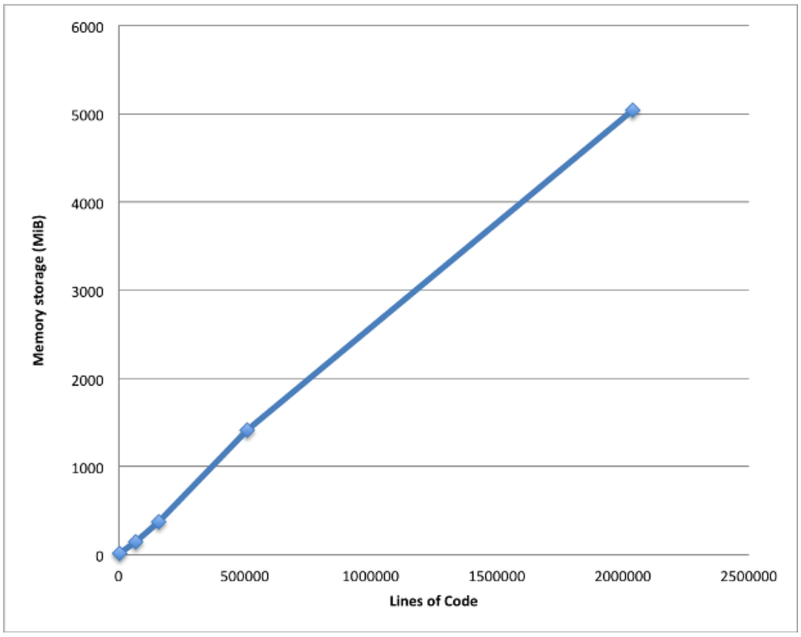


Figure 4.10: Lines of Code vs. Memory storage

4.7.4 Comparison with other systems

Unfortunately, we could not find a working implementation for recent system such as Codequest and JTL. However, the authors reported absolute times of general source code queries executed on their system, which we briefly compare with. The reader should note that it is difficult to make a fair comparison since the queries were executed on slower machines than we used for our experiments. However, our system stores more information about the source code of programs, which also impacts performance. For example, CodeQuest, JTL and JQuery do not support method bodies, generics or types of expressions.

JTL The authors reported execution times between *10s* and *18s* for code exploration queries carried out on *12,000* random classes taken from the Java standard library. However, it is unclear whether the experiment were performed with a cold or warm system.

We reported times of 0s to 12s running on a *cold* system and 0s to 3s with a *warm* system by running queries on a larger data set (a Corpus of over *13,000* Java source files and 2.04 million lines of code). Note that our setup is given an advantage because the JTL experiments were reported on a 3GHz Pentium 4 processor with 3GB of memory whereas we used a more modern machine (MacBook Pro with 2.7GHz Intel Core i7 with 8GB of memory).

Codequest The authors ran various queries on different database systems. We compare with the fastest times they report, which were found on a quad Xeon 3.2Ghz CPU with 4GB of memory and using XSB (a state of the art deductive database). The authors reported times between *0.1s* and *100s*. The slowest query is looking for methods overriding a parent method defined as `abstract`. In the Codequest system, the overriding relation between methods is compted on the fly. In our system, the override relation is already pre-computed as an *overlay* on the AST.

JQuery We created three queries in JQuery (not to be confused with the jQuery JavaScript library): finding recursive methods, classes declaring public fields and overloaded methods as shown in Figure 4.11. We tested them on the *ivy* program (68K LoC) because JQuery failed indexing larger projects such as *Hsqldb*. This deficiency was already reported by other work [45, 66]. All queries executed in a couple of seconds except overloaded methoded which took over 30s on our MacBook machine.

```
// 1. recursive methods
method(?C,?M), calls(?M,?M,?L);

// 2. classes declaring public fields
class(?C), field(?C,?F), modifier(?F, public);

// 3. overloaded methods
method(?C1, ?M1), method(?C2, ?M2),
likeThis(?M1,?M2)
```

Figure 4.11: Test queries for JQuery

4.8 Related work

In this section, we review existing source code querying systems and highlight what information about the program is lost and the consequential restrictions on source code queries.

Source code querying systems can be classified in two categories:

1. **software**: query the source code of one piece of software in particular.

2. **repository**: query the source code of multiple projects at the same time.

We provide a detailed comparison of software-level source code querying languages in a previous paper [129]. We evaluated whether their expressiveness suffices for language design research. To this end, we studied several use cases of recent Java features and their design issues – investigating their expressibility as queries in the source code querying languages we examined. For example, one use case asks to query for covariant array assignments. We found that only two query languages provide the minimal features required to express all our use cases.

4.8.1 Software querying systems

Java Tools Language (JTL) is a logic-paradigm query language to select Java elements in a code base and compose data-flow queries [45]. The implementation analyses Java bytecode classes. Since JTL is based on bytecode analysis rather than source code analysis, several queries are inherently restricted. For example, Java generics are compiled to casts during the bytecode translation.

Browse-By-Query (BBQ) reads Java bytecode files and creates a database representing only classes, method calls, fields, field references, string constants and string constant references [2]. This database can then be interrogated through English-like queries.

SOUL is a logic-paradigm query language [50]. It contains an extensive predicate library called *CAVA* that matches queries against AST nodes of a Java program generated by the *Eclipse JDT*.

JQuery is a logic-paradigm query language built on top of the logic programming language *TyRuBa* [70]. It analyses the AST of a Java program by making calls to the Eclipse JDT. JQuery includes a library of predicates that lets the user query Java elements and their relationships.

Codequest is a source code querying tool combining Datalog as a query language and a relational database to store source code facts [66]. Datalog queries are translated to optimised SQL statements. The authors report that this approach scales better compared to JQuery. However, only a subset of Java constructs are supported and types of expressions are not available. For example, one could not locate assignments to local variables of a certain type.

.QL is an object-oriented query language. It enables programmers to query Java source code with queries that resemble SQL [49]. This design choice is motivated as reducing the learning curve for developers. In addition, the authors argue that object-orientation provides the structure necessary for building reusable queries. A commercial implementation is available, called *SemmleCode*, which includes an editor and various code transformation.

Jackpot is a module for the *NetBeans IDE* for querying and transforming Java source files [11]. Jackpot lets the user query the AST of a Java program by means of a template representing the source code to match. However, Jackpot does not provide any type information for AST nodes

PMD is a Java source code analyser that identifies bugs or potential anomalies including dead code, duplicated code or overcomplicated expressions [21]. One can also compose custom rules via an *XPath* expression that queries the AST of the program. However, types of expressions are not available.

4.8.2 Repository querying systems

Sourcerer is an infrastructure for performing source code analysis on open source projects [35]. It consists of a repository of Java projects downloaded from Apache, Java.net, Google Code and Sourceforge. Source code facts of the projects in the repository are extracted and stored in a relational database that can be queried. Information available include program *entities* such as classes or local variable declarations. It also includes *relations* between entities such as `CONTAINS` (e.g. a class containing a method) or `OVERRIDES` (e.g. a method overrides a method).

Sourcerer focuses on storing structural information. For example, statements within a method are not available. In addition, queries involving types of expressions are not supported.

Boa is an infrastructure for performing metadata mining on open source software repositories [54]. It indexes metadata information about projects, authors, revision history and files. However, it store no source code facts. The infrastructure provides a query language to mine information about the repositories. For example, one can query the repositories to find out the 10 most used programming languages. Currently Boa indexes more than a million projects from Github, Google Code and Sourceforge. Overall scalability is achieved by using a Hadoop as a map-reduce framework to process the data in parallel.

4.9 Conclusion

This chapter presented the design and implementation of a source code querying system using a *graph database* that stores full source-code detail and scales to program with million lines of code.

First, we described a model for Java source code based on the *graph data model.* We introduced the concept of *overlays*, allowing queries to be posed at a mixture of syntax-tree, type, control-flow-graph and dataflow levels (Section 4.4).

Next, we showed that a graph query language can naturally express queries over source code by showing examples of code exploration and language research queries written in CYPHER (Section 4.5).

Finally, we evaluated the performance of our systems over programs of various size (Section 4.7). Our results confirmed that graph databases provide an efficient implementation for querying programs with million lines of code while also storing full source-code detail.

Chapter 5

Incompatibility-finding via type-checking

In the previous chapter, we described how a source-code querying system can help developers locate parts of code relevant to programming language evolution. We argued that the data structures generated by a compiler can be exposed and queried to identify relevant code patterns. We presented several examples for Java. Conveniently, the Java compiler makes available type information in these data structures at compile time. This type information is useful to express sophisticated code patterns (i.e. beyond simple text match) which is required to locate non-trivial incompatible changes.

This chapter turns to dynamically-typed languages which typically do not have type information available at compile time. In fact, such information about a program may only be available during the program's execution. Consequently, it is difficult for programmers to understand and locate the source of problems caused by incompatible changes.

To tackle this issue, this chapter proposes a system which uses type annotations that are checked during the program's execution and reports errors. The reported errors provide useful insights to the programer about the location of code affected by changes. We show how this system can help programmer understand incompatible changes when a program is executed with a different language version by using the type annotations as contracts. We present this system for Python but much of this work is language-agnostic. The implementation of our system has two important characteristics. First, we design a rich language to express type annotations which can cater for various common use cases with code written using a dynamically-typed language. Second, the checking and reporting mechanism monitors the input and output behaviour of function definitions at runtime. We implemented the type-checking mechanism as a lightweight library which dynamically enhances function definitions with type-checking capabilities to facilitate the integration of our system into existing projects. Our system makes use of local inference to implement certain features of the type system.

5.1 Introduction

Python is a dynamically-typed programming language that is popular because of its simple syntax which makes it very accessible. Indeed, Python has been ranked in the top ten most-used programming languages of the TIOBE index (monthly update of popular language use) for the past ten years [117]. Python is used for many applications such as scientific calculations, building large websites or just scripting [58].

As we mentioned in Chapter 3, Python 3 introduced various behavioural incompatibilities with Python 2. Specifically, these incompatibilities involve different objects (i.e. with different types) returned by Python operations and functions. We revisit these issues in more detail later.

This chapter develops a type-checking system for Python using type annotations provided by the programmer. The type annotations are checked during the program's execution to help programmers locate errors. In our system called *pytypedecl*, programmers annotate the arguments and result of functions with the types that they expect. The type-checker can then report a list of precise type errors when the type assertions are violated. In other words, *pytypedecl* lets programmers adopt a design-by-contract style in Python using runtime checks [84]. Our approach differs from a full blame-tracking approach because violations are only checked at the entrance and exit of a particular function. As a result, our system has a smaller overhead. Type assertions may fail for various reasons, for example, if the programmer introduces a mistake in the code or if the program is run with a different version of Python which introduces type incompatibilities. The programmer can then interactively fix the function definition that is the source of the problem. Figure 5.1 illustrates our system using a program written in Python 2 and later executed using Python 3.

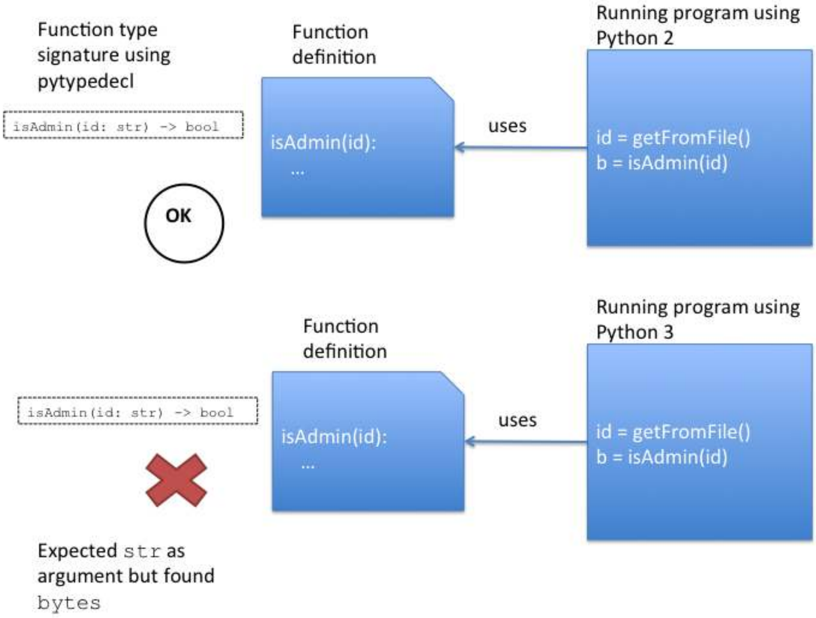


Figure 5.1: Incompatibility-finding via type-checking

There have been many attempts to provide enhanced type-checking for dynamic languages such as gradual typing and type inference [78, 56, 121, 60]. However, most systems are typically difficult to integrate in an existing developer infrastructure within a large company. For example, these systems are typically implemented as a separate independent tool or an extension of the language. This makes them problematic to integrate into a build pipeline or company infrastructure constraints. Some systems may also require modifications to existing source code files to add type annotations.

Pytypedecl is designed as a Python package. Programmers annotate functions and these type annotations are specified in separate files. In addition, pytypedecl provides a rich language to express types to be able to check various constraints. Pytypedecl supports exceptions, union types, intersection types, noneable types, interface types and parametric polymorphism. We will discuss these features in the next sections.

The rest of this chapter is structured as follows. First, we present the design space for our proposed system and describe its formalisation in an iterative manner. Next, we describe the implementation of our system. Then, we evaluate pytypedecl's features on a set of Python programs. We then provide further detail about the behavioural incompatibilities in Python 2 and Python 3 and develop supporting examples of how pytypedecl can help programmers understand incompatibilities introduced during programming language evolution. Finally, we present related work and conclusions.

5.2 Design space

This section discusses the design space for developing a tool for Python which helps developers locate errors resulting from incompatible types. First, we discuss type-checking approaches and the trade-offs of implementing them for Python. Next, we review the expressiveness of type declarations of other programming languages in comparison to Python. Finally, we argue for a new approach which provides additional type declarations than what Python offers together with type-checking at boundaries of function.

5.2.1 Type-checking approaches

In this subsection we discuss existing type-checking approaches that are related to our work. Table 5.2 summaries the trade-off of existing approaches in Python.

Static type-checking Static type-checking verifies that a value conforms to a type at compile time, before the program is run. As a result, programmers can find common programming errors earlier. For example, the following code in Java is caught as a compile error because it is forbidden to subtract an `int` from a `String`:

```
String hello = "hello";
int number = 1;
System.out.println(hello - number);
// Operator "-" cannot be applied to java.lang.String and int
```

Adding static type-checking to Python is difficult because many idioms rely on Python's dynamic features including introspection (e.g. `hasattr`) and runtime code evaluation using `eval` and `exec` [111]. The use of these features generate values whose types can only be determined when the program is run. Additionally, many common code pattern require additional complexity in the type-system to include flow and context-sensitive checking. For example, a variable's type in Python can differ along different paths such as inside an `if` block. In addition, because Python does not support static type-checking out of the box, implementing it requires developing and maintaining a separate system which parses and checks Python code.

Dynamic type-checking Dynamic type-checking verifies type safety at run time (i.e. when the program is run) [122]. One benefit of this approach is that it allows rapid prototyping as the programmer does not have to think in terms of type constraints. However, as a consequence, dynamic type-checking can cause a program to fail at runtime. For example, the following code in Python will only fail when the program is executed:

```
"hello" - 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'int' and 'str'
```

However, such errors can often occur later in the execution of the program, long after the operation that caused it. Consider the following Python code:

```
def getPrice():
  return 10000, # accidental trailing comma, creates a tuple

def halfPrice(price):
  return price / 2

price = getPrice()
halfPrice(price)
```

Running this program results in the following error at runtime:

```
Traceback (most recent call last):
  File "ex1.py", line 9, in <module>
    halfPrice(price)
  File "ex1.py", line 5, in halfPrice
    return price / 2
TypeError: unsupported operand type(s) for /: 'tuple' and 'int'
```

The error only occurs at line 5, after the `getPrice` function was called and during the execution of the function `halfPrice`. A static type-checking system would report before the program is run that `getPrice` should return an `int` but returns a `tuple` instead.

Mixed typing The presence of static type-checking and dynamic type-checking are not mutually exclusive. For example, Java supports covariant arrays. For example, type safety of an assignment to a component of the array is only checked at run-time to ensure that the value being assigned is appropriate [65].

```
Object[] objects = new String[5];
objects[0] = 1;
//java.lang.ArrayStoreException: java.lang.Integer
```

Many approaches have been proposed in the literature to strike a balance between the benefits of static type-checking and dynamic type-checking. These approaches are referred to as *mixed typing* [78]. For example, Fagan et al. proposed *soft-typing* [56]. They key idea is that runtime checks are inserted in the program to log parts of the program that may contain potential errors using a static-checking approach.

Thatte et al. proposed *quasi-static typing* [121]. Part of the program annotated with types are statically checked and the other parts dynamically checked. However, later work showed that this mechanism does not statically catch all type errors in completely annotated programs because an unknown type was treated as both the top and bottom of the subtype lattice. This work was later extended by Siek et al [116]. and is known as gradual typing, which we discuss next.

Gradual typing Gradual typing lets programmers use static type-checking for certain parts of their code and dynamic type-checking for other parts of their code. As a result they can catch errors earlier than in a pure dynamic type-checking system but they can also benefit from the flexibility provided by dynamic type-checking. When values flow from a statically-checked part of the code into a dynamically-checked part of the code and vice versa, gradual typing inserts dynamic type casts to ensure consistency. These type casts are used to localise the source of an error. The process of localising the source of the error is called *blame tracking.* Consider the previous example using explicit Python 3 type annotations syntax:

```
def getPrice():
  return 10000, # oops trailing comma

# Python 3 type annotations syntax
def halfPrice(price: int) -> int:
  return price / 2

price = getPrice()
halfPrice(price)
```

The value `price` flows into the function `halfPrice` which is statically typed and expects an `int` as argument. Instead of producing the latent error shown previously where the division operator is not supported for a tuple and int, blame tracking immediately reports that a value of an invalid type was passed as argument to `halfPrice`.

Type inference A related topic to type-checking is that of type inference. The general idea is that an inference algorithm can automatically deduce types for values based on their use. Such analysis can be performed both without running the program (static type inference [60]) and by running the program (dynamic type inference [32]). Approaches to add static type-inference to Python include using the Cartesian Product Algorithm (CPA) [114]. This was shown to work on a subset of Python's language features using small isolated programs which do not call external libraries. Recent experiments have shown that inference using CPA can take up to 7s for a program of 1000LoC [79]. Today's programs are in the region of hundreds of thousands of lines of code. In addition, CPA is inaccurate in the presence of dynamic features which are idiomatic in Python as discussed previously. We present an alternative type inference technique in the next chapter.

Type-checking Approach	Pros	Cons
Static	• Catch errors before program is run	• Complex type system • Does not deal with dynamic language features • Replacement front-end
Dynamic	• Rapid prototyping • Avoid infeasible paths	• Latent errors
Mixed	• Catch some errors before program is run • Allows use of dynamic language features	• More complex type system • Replacement front-end

Figure 5.2: Type-checking approaches for Python

5.2.2 Type declarations

This section briefly presents an overview of the design space for expressing type annotations in Python by comparing with other programming languages. It then summarise a list of type annotation features which are implemented in our system and presented in subsequent sections.

Python 3 already provides syntactic support for describing function signatures. Programmers can add annotations around function parameters and return values. For example in Python 3, one can describe a function `get` which has a parameter `i` of type `int` and returns a `str`:

```
def get(i: int) -> str:
    ...
```

However, these signatures in Python 3 are used purely for documentation. Moreover, the expressiveness of this syntax is limited. For instance, exceptions and overloaded signatures are not supported. The lack of support for more sophisticated type annotations is part of our motivation for designing a more expressive type declaration language, which we develop in the next sections. We selected a set of features based on common use cases that can be expressed in Scala, Typescript, the type annotation language accepted by the Closure compiler (a tool maintained by Google which support type annotations for JavaScript), Java and C#. Figure 5.3 compares the type features supported in other programming languages and what Python 3 currently offers. We then describe each feature and how they are incorporated into pytypedecl's type declaration language.

Function signature Functions can be given a signature following the Python 3 function annotation convention. However, Python 3's syntax does not allow users to specify a list of possible exceptions that the function may raise.

Feature	**Scala**	**Java**	**C#**	**Closure compiler**	**Typescript**	**Python 3**
Function signature	✓	✓	✓	✓	✓	limited
Overloading	✓	✓	✓	✓	✓	×
Interfaces	✓	✓	✓	✓	✓	×
Noneable types	✓	✓	✓	✓	✓	×
Union types	×	×	×	✓	✓	×
Intersection types	✓	✓	×	×	×	×
Generics	✓	✓	✓	✓	✓	×

Figure 5.3: Type declaration design space

Overloading A function is allowed to have multiple different signatures. This is not supported in the Python 3 function annotation syntax.

Interfaces An interface describes a set of operations that a type must support. It can be seen as defining a structural type with an additional name to refer to that structure. Interfaces can also be multiply inherited, as in Java and traits in Scala.

Noneable types A noneable type can represent the range of values for its value type as well as an additional `None` special value in Python. For the rest of the chapter, we will use the syntax of trailing `?` to indicate that a type is noneable. For example, a parameter of type `int?` can accept a `int` or a `None` value.

Union types It is sometime convenient to indicate that a type can hold values from a type or another one. Union types express this idea. For example `int | float` (the operator `|` here is used for denoting disjunction) indicates that a value may be an `int` *or* a `float`. There is no limit to the number of types of a union. Note that a noneable type is shorthand for the union of a given type and the built-in Python type `NoneType`. Hence `int?` is equivalent to `int | NoneType`.

Intersection types An intersection type describe a type that belongs to all values of a list of types [110]. For example, the intersection type `Readable & Writable` (the operator `&` here is used for denoting intersection) describes a type that supports the operations `open`, `close`, `read` and `write`. However, the intersection type `int & float` would not be very useful because there is no Python value that is both an `int` and a `float`.

Generics Types can be parameterised with a set of type arguments, similarly to Java generics. For example, `list[str]` describes a list that only contains strings, `dict[int, str]` describes a dictionary of keys of type `int` and values of type `str`.

We now describe how our tool`pytypedecl` fits in the design space.

5.2.3 Discussion

Our tool `pytypedecl` verifies type signatures in Python code by type-checking function boundaries at runtime. This design choice was influence by three requirements:

- *Precision*: catch latent errors
- *Practicability*: naturally integrates in existing Python development workflow
- *Simplicity*: the type system should feel familiar to Python developers and deliver understandable diagnostics

Static-type checking is too strict as it does not allow the use of dynamic language features such as `eval`, `exec` and `hasattr` which are common idioms in Python[111]. Moreover, many common code pattern require additional complexity in the type-system to include flow and context sensitivity checking which are unfamiliar to Python developers. An extreme example of how a complex static type-checking system can cause harms for its users is that of Scala where compile errors are often problematic to understand [1].

In addition, implementing static-type checking for Python requires developing and maintaining separate tools to parse and type-check annotated programs which restrict its adoption in industrial settings. In fact, this was pointed out as a key requirement in the design of Coverity to enable its adoption: "tools could often require painful, time-consuming customization or integration efforts for teams that used anything but the most common technologies such as plain Makefiles with a single target, antfiles" [38, 6].

The downside of a full dynamic-typing approach is that they do not catch latent errors. Mixed typing approaches combine both approaches but as a result also share similar downsides to static-type checking including a complex type system and require separate tools.

We propose a system called *pytypedecl* inspired from contract systems which catches latent errors earlier than dynamic-type checking and facilitate adoption by naturally integrating in a developer's workflow. Our system consists of a runtime type checking system that checks runtime tags of Python values against statically declared types which are specified on functions. We designed a type declaration language to specify these types as presented in the previous section.

Our system is implemented as a library which means it only needs to be imported in a Python module that requires type-checking without interfering with the developer workflow. An analysis of Python projects that we present in Section 5.8.2 shows that Python developers manually implement similar type-checks to act as unit tests which require long-term code maintenance. This strengthen the need for our system.

5.3 Basic system

This section presents a basic system which only considers a first-order version of Python (i.e. no higher-order functions and generators), Python built-in types, noneable, union, intersection and interface types but no parametric or ad-hoc polymorphism. In the next sections we successively enrich the type system. Firstly, we include qualified parametric data types for built-in types and user-defined class types. Secondly we include parametric polymorphism for functions. We then include bounded quantification. Finally, we also include overloading and discuss support for more sophisticated Python features including generators and first-class functions.

5.3.1 Type annotations

We start by introducing the syntax for the static type annotations used in the basic system. We then develop the type checking rules for programs that use these types.

All data in Python is represented by objects. Every object has a type, a value and one or more bases (i.e. super-classes) from which it inherits. An object's type determines the operations supported by an object and also defines the possible values for that object's type. A type can be a built-in type such as an `int`, `float` or `object` but also a user-defined class type such as `Fruit`. A value defines the *content* of the object and may include attributes. To avoid confusion with the static types defined in pytypedecl we will refer to an object type in Python as a *type tag*. The type tag of an object can be returned using the built-in Python function `type()`. Type tags can only be user-defined classes or Python built-in types such as `int`, `float`, `list` etc. In pytypedecl, we will refer to these elementary type tags as *ground types*.

Figure 5.4 presents the grammar for the types available in the basic system.

```
identifier ::= (letter|”_”)(letter|digit|”_”)*
op ∈ Operations ::= identifier

b ∈ BaseType ::= int | float | list | ...
c ∈ ClassType ::= identifier
g ∈ GroundType ::= BaseType | ClassType
ex ∈ ExceptionType ::= Classtype

t ∈ Type ::= b | c | Unknown |
             Noneable[t] |
             Union[$t_1$, ..., $t_p$] | Intersection[$t_1$, ..., $t_q$] |
             Interface[identifier, {$op_1$, ..., $op_s$}]
```

Figure 5.4: Static types for basic system

The type annotations in pytypedecl includes Python built-in types as part of the `BaseType` set. The set `ClassType` refers to user-defined classes and the set `ExceptionType` to exceptions. We will use the meta-variables `b`, `c`, `g` and `ex` to refer respectively to base types, class types, ground types and exception types.

As shown in Figure 5.4, the set `Type` represents all the possible type annotations in pytypedecl. We will use the metavariable `t` to refer to a type in pytypedecl. In addition to base types and class types it also includes the type `Unknown` to specify that the type of a value is not known (i.e. it is dynamic) and various type constructors:

- A `Noneable` type describes a type that can be either the built-in type `NoneType` or another type. It is analogous to nullable types in other systems [115].

- A `Union` type represents a union of several types. For example `Union[int, float]` indicates that a value may be an `int` or a `float` [68].

- An `Intersection` type represents a type that belongs to all values of a list of types.

- An `Interface` type describes a set of operations that a type must support. It can be seen as defining a structural type with an additional name to conveniently refer to that structure [80]. Interfaces can also be multiply inherited. For conciseness purposes in our system an `Interface` contains not only the operations it defines explicitly but also the operations it inherits implicitly from its super interfaces.

We now discuss where these type annotations can appear.

5.3.2 Function type signatures

In pytypedecl, types appear as part of a summary function signature in a separate file for a Python function. A summary function signature consists of a type annotation for each parameter, a type annotation for the return type and an optional list of exception types that the function may raise. Figure 5.5 presents the syntax for a summary function signature.

```
def f($p_1$ : $t_1$, ... $p_n$ : $t_m$) $\rightarrow$ $t_0$ raises $ex_1$, ..., $ex_n$
```

Figure 5.5: Function type signature syntax

The absence of a type annotation for a parameter or return type implies that the type is `Unknown`. A value of type `Unknown` is not type-checked by pytypedecl. In the subsequent sections, we will refer to the summary function signature of a given function as its *formal definition*. We now describe how the summary function signatures are used to type-check Python programs.

5.3.3 Type checking

Pytypedecl is designed to only check types at function boundaries. When a function is called, pytypedecl looks up its associated summary function signature. Pytypedecl then checks that the value of each argument of the function call instance is compatible with the respective static type annotation declared in the function signature.

After that, the body of the function is executed. The result or a possible exception raised is captured and checked that it is consistent with the function signature. Figure 5.6 summaries these steps.

```
check(f, [$a_1$, ..., $a_n$], [$t_1$, ..., $t_n$], $t_0$, [$ex_1$, ..., $ex_m$]):
    $\forall i \in [1..n] : assert\ isCompatible(a_i, t_i)$
    try:
        $r_a := f(a_1, ..., a_n)$
        $assert\ isCompatible(r_a, t_0)$
        $return\ r_a$
    except Exception as ex:
        $assert\ ex \in [ex_1, ..., ex_m]$
        $raise\ ex$
```

Figure 5.6: Type-checking function boundaries

We outline the type checking mechanism with the routine `check`, which takes as first argument a reference to the function associated with the function call instance, a list of actual arguments, a list of pytypedecl types for each argument, a result type and a list of possible exceptions. If the function returns a value after execution, we check its type is compatible with the result type declared in the function signature. If it raises an exception, we check that the exception was declared in the list of exceptions.

A further design option is whether to also check the arguments on function exit as well. This is useful to check whether the arguments are mutated with conflicting values within the body of a function. For example, in the following code, the fact that the parameter `id` is an `int` and the function body assigns a `str` to it is not captured with the rules outlined in Figure 5.6:

```
def get(id: int) -> str
    if id == 0:
        id = "admin"
    else:
        id = "user"
    return id
```

Note that, this is less important here because changes to `id` do not affect the caller. However, if `id` was a list of numbers then an element inside the list could be updated to a string by accident. Checking arguments on function exit would capture these kind of problems.

Figure 5.7 provides the definition of `isCompatible` that checks whether a value is compatible with its formal definition. The parameter `a` refers to an actual value and `t` to a type available in pytypedecl as defined in Figure 5.4. We use a special syntax to apply pattern matching on the parameters of the `isCompatible` function using a `case` clause.

In a nutshell, built-in and named class types are simply checked using Python's `instance` built-in function. The `instance` function is a predicate which tells whether a value is a subtype a given type tag. To satisfy the union of several types, we only need to check if the value is compatible with *at least one* type in the list of types encapsulated by the union. To contrast, to satisfy the intersection of several types, we need to check that the value is compatible with *all* types in the intersection list. Finally, to check if a value is compatible with an interface, we check if the value supports all operations listed in the interface type. Interfaces are used to support structural typing so the name of the interface is ignored in the type-checking process. The built-in Python function `dir` is used to return the list of operations for a given value.

```
isCompatible(a, t):
    $| case\ \_Unknown = true$
    $| case\ a\ BaseType = isinstance(a, t)$
    $| case\ a\ ClassType = isinstance(a, t)$
    $| case\ a\ Union[t_1, ..., t_p] = \exists t' \in [t_1, ..., t_p] : isCompatible(a, t')$
    $| case\ a\ Intersection[t_1, ..., t_p] = \forall t' \in [t_1, ..., t_p] : isCompatible(a, t')$
    $| case\ a\ Interface[name, (op_1, ..., op_s)] = \forall op \in [op_1, ..., op_s] : op \in dir(a)$
```

Figure 5.7: Type-checking a function call

5.4 Qualified parametric data types

We now consider an extended system that includes parametric data types. Python supports several built-in heterogenous containers such as `list`, `set` and `dict`. In this section, we extend our typesystem with homogenous containers as well as parametric user-defined classes. The remainder of this section uses the following terminology:

- Data types such as `list`, `set` and `dict` are *parametric type constructors.*
- We call `list[int]` a type instance and `int` is its (singleton) tuple of type arguments. We will refer to *qualified parametric data types* as a parametric type constructor that is given an explicit tuple of type arguments.

- A parametric user-defined class is a user-defined class that is parameterised over types. For example `Pair[int, float]` is a parametric user-defined class where `Pair` is a user-defined class, and `int, float` is its tuple of type arguments.

We consider type variables (e.g. `list[T]` where `T` is a type variable) appearing in function signatures in the next section.

5.4.1 Type annotations

Pytypedecl allows built-in containers such as `list`, `tuple`, `set` and `dict` to be parameterised as well as user-defined class types. Figure 5.8 extends the type annotations specification from Figure 5.5 to support type annotations for qualified parametric data types.

```
t ::= ...  |
     list[t] | tuple[t_1, ..., t_p] | set[t] | dict[t_1, t_2] |
     ClassType[t_1, ..., t_u]
```

Figure 5.8: Extended grammar with qualified parametric data types

5.4.2 Type checking

We now present the type checking mechanism for qualified parametric data types. We consider *container integrity* which is responsible for checking whether a parameterised type holds values that are compatible with its type arguments. Parametric types with a single type argument such as `list` and `set` can be simply checked by iterating through all the values of the container. In Python, iteration is typically implemented by providing an implementation for an `iterator` object. Tuples are type-checked by checking whether each element is compatible with its associated type argument. Note that since types are only checked on entry of functions (and mutations inside the function body are not tracked), it is convenient to treat parametric data types covariantly by default (e.g. a programmer might want to pass a `list[Apple]` where a `list[Fruit]` is expected provided that `Apple` is a subtype of `Fruit`).

User-defined classes require a different type-checking mechanism because they can have an arbitrary number of type arguments. In our system, the user is responsible for providing an iterator which extracts a tuple of values. Each value is associated to a type argument. This iterator can be used to check whether the values are compatible with the associated type arguments. Figure 5.9 extends the definition of the `isCompatible` routine presented in Figure 5.7 to support parametric qualified types. Note that for user-defined classes, we use the meta-variable `c` which ranges over all user-defined classes to identify parametric user-defined classes.

5.4.3 Discussion

Another form of type checking that pytypedecl does not consider is *structure integrity* which is responsible for checking whether a type instance is well formed in a similar way to an algebraic data type. For example, to check whether an AST is correctly constructed (e.g. only supported arithmetic operations appear in an expression). Such checking can

```
isCompatible(a, t):
   | ...

   | $case\ a\ list[t] = isinstance(a, list) \wedge \forall e \in a : isCompatible(e, t)$
   | $case\ a\ set[t] = isinstance(a, set) \wedge \forall e \in a : isCompatible(e, t)$
   | $case\ a\ tuple[t_1, ..., t_p] = isinstance(a, tuple) \wedge$
                                   $\forall i \in \{1, .., p\} : isCompatible(a[i], t_i)$
   | $case\ a\ dict[t_1, t_2] = isinstance(a, dict) \wedge$
                              $\forall (k, v) \in a : isCompatible(k, t_1) \wedge isCompatible(v, t_2)$
   | $case\ a\ c[t_1, ..., t_u] = isinstance(a, c) \wedge$
                              $\forall (e_1, ..., e_u) \in a :$
                              $isCompatible(e_1, t_1) \wedge ... \wedge isCompatible(e_u, t_u)$
```

Figure 5.9: Type-checking qualified parametric types

be provided by the iterator implemented by the user or as the data structure is created via its constructor.

5.5 Parametric polymorphism

We now extend pytypedecl by introducing parametric polymorphism for functions. For example, in Java, the signature below describes a generic method named `identity` which is parameterised with the type variable `T`:

```
<T> T identity(T e);
```

The type variable is used to specify that the type of the parameter `e` and the result type of the method have to be identical. In a more mathematical notation, the type of identity is $\forall$ `a.a` $\rightarrow$ `a`, where `a` is a type variable.

We extend the type system of pytypedecl with a similar mechanism. Our implementation makes use of type inference via unification to verify the constraints placed by the type variables. We discuss how to extend this mechanism with bounded quantification (i.e. specifying a type bound on the type variables) in the next section. The rest of this section starts by extending the type declarations in pytypedecl with support for type variables. It then outlines the type checking mechanism to support parametric polymorphism.

5.5.1 Type annotations

We extend the grammar of the type annotations presented in Figure 5.8 to include type variables. Figure 5.10 presents the update to the grammar. For simplicity, type variables can only be a single upper-case letter identifier from `A` to `Z` included. The meta-variable `v` refers to the set of all type variables. We add a new type `BoundedTypeVariable` to represent a type variable in the type declaration language. It is included in the set of possible pytypedecl types denoted by the meta-variable `t`. We will consider that all types variables have a default bound that is `object` for now. I.e. in a more mathematical notation, $\forall$ `a` $\leq$ `object`, where `a` is a type variable and `object` is its default bound. In the next section, we extend the system with more specific bounds.

```
upper ::= 'A' | ... | 'Z'
v ∈ TypeVariable ::= upper
t ::= ... | BoundedTypeVariable[v, t]
```

Figure 5.10: Type variables

5.5.2 Function type signatures

We extend the syntax for the summary function type signatures in pytypedecl to support type variables. In a similar syntax to Scala, an optional list of type variables can be declared before the function name as shown in Figure 5.11.

```
def f[$v_1$, ..., $v_n$]($p_1$ : $t_1$, ... $p_n$ : $t_m$) → $t_0$ raises $ex_1$, ..., $ex_n$
```

Figure 5.11: Function type signature supporting type variables

We now discuss the implementation of the type checking mechanism to support type variables.

5.5.3 Type checking

Consider the following function signature in pytypedecl

```
findItem[T](a: list[T], b: T) -> bool
```

Given a call instance findItem(e_1, e_2), the type-checking process needs to infer a ground type for the type variable T that is consistent with the type tags of the parameters discovered at runtime. If no ground type can be inferred then the call instance does not type-check.

We propose a semantic unification algorithm for inferring the type variables. Before we outline the algorithm, we describe an example to motivate its development. Suppose a call instance to findItem where the first parameter e_1 has type list[Apple] (i.e. all the values inside the list are type Apple) and the second parameter e_2 is type Apple. The type-checking process would succeed because T is unified to the type Apple. However, suppose another call instance to findItem received for the second parameter a value of type Banana instead. Then type-checking would fail with a simple equational unification because Apple and Banana are different types. Nonetheless, Fruit is also an acceptable type for T. To solve this problem, pytypedecl unifies a type variable by calculating the least common upper bound between possible types for it. We choose to allow the least upper bound except in the case when the least upper bound is object. If we did not make this exception, all type variables could be inferred to object which defeats type-checking.

Figure 5.12 presents the extension to the pytypedecl type checking mechanism to support parametric polymorphism. The unification algorithm used for inference of type variables has four components:

1. The set v of possible type variables as defined in Figure 5.10.
2. A set of elements that can be associated to type variables. In pytypedecl case, elements are the set g of ground types as defined in Figure 5.4.

3. A type variable mapping `TVM`: $v \rightarrow g$ assigning a ground type to each type variable.

4. An equivalence relation on `g` indicating when elements are considered equal. As described previously, pytypedecl type-checks a type variable for two different ground types when they have a common least upper bound that is not `object`.

The updated `check` routine presented in Figure 5.12 initialises a fresh type variable mapping `TVM` for each call instance being type checked. `TVM` is then shared with the `isCompatible` routine and updated via side-effect. Figure 5.12 also shows the updated implementation for `isCompatible`. We define a separate routine `leastUpperBound` to find the least upper bound between two ground types. It uses three helper functions:

- a function `getSuperTypes` which transitively returns all super-types of a given ground type including itself.

- a function `getMostSpecific` which given a list of ground types returns the most specific type that is not `object`. If there are none it fails.

- a function `issubclass` (which is built-in function in Python) which takes two ground types as argument g_1 and g_2 and returns true if g_1 is a direct or indirect subclass of g_2.

5.6 Bounded quantification

So far all type variables had an implicit default bound of `object`. We now add bounded quantification [105] to pytypedecl. A bound restricts a type variable to only range over subtypes of a given type. For example, in Java, the following generic method restricts the type variable `T` to only range over subtypes of `Message`.

```
<T extends Message> T combine(T m1, T m2);
```

Pytypedecl supports type bounds over ground types which are restricted to be built-in Python types and class types as presented in Section 5.3. As a result, pytypedecl does not allow parametric types or type variables to appear as a bound. This restriction allows the inference rules to remain simple. This section starts by extending the syntax for function type signatures in pytypedecl to support type bounds. We then extend the type-checking mechanism to support it.

5.6.1 Type annotations

Pytypedecl uses the operator `<:` to express a sub-typing constraint for a type variable. The left-hand side is a type variable and the right-hand side is a ground type. If no type bound is specified, by default pytypedecl assumes it is `object`. Figure 5.13 adds syntactic support for type bounds in pytypedecl.

```
check(f, [$a_1$, ..., $a_n$], [$t_1$, ..., $t_n$], $t_0$, [$ex_1$, ..., $ex_m$]):
  TVM := {}
  $\forall i \in [1..n] : assert\ \ isCompatible(a_i, t_i,$ TVM$)$
  try:
    $r_a := f(a_1, ..., a_n)$
    $assert\ \ isCompatible(r_a, t_0,$ TVM$)$
    $return\ r_a$
  except Exception as ex:
    $assert\ \ ex \in [ex_1, ..., ex_m]$
    $raise\ ex$
```

```
isCompatible(a, t, TVM):
  $\mid \_\ Unknown$ TVM $= True$
  $\mid a\ b$ TVM $= isinstance(a, b)$
  $\mid a\ c$ TVM $= isinstance(a, c)$
  $\mid a\ Union[t_1, ..., t_p]$ TVM $= \exists t_i \in [t_1, ..., t_p] : isCompatible(a, t_i, TVM)$
  $\mid a\ Intersection[t_1, ..., t_p]$ TVM $= \forall t_i \in [t_1, ..., t_p] : isCompatible(a, t_i, TVM)$
  $\mid a\ Interface[name, (op_1, ..., op_s)]$ TVM $= \forall op \in [op_1, ..., op_s] : op \in dir(a)$
  $\mid a\ list[t]$ TVM $= isinstance(a, list) \wedge \forall e_i \in a : isCompatible(e_i, t, TVM)$
  $\mid a\ set[t]$ TVM $= isinstance(a, set) \wedge \forall e_i \in a : isCompatible(e_i, t, TVM)$
  $\mid a\ tuple[t_1, ..., t_p]$ TVM $= isinstance(a, tuple) \wedge \forall i \in \{1, .., p\} : isCompatible(a[i], t_i, TVM)$
  $\mid a\ dict[t_1, t_2]$ TVM $= isinstance(a, dict) \wedge$
   $\forall (k_i, v_j) \in a : isCompatible(k_i, t_1, TVM) \wedge isCompatible(v_j, t_2, TVM)$
  $\mid a\ c[t_1, ..., t_u]$ TVM $= isinstance(a, c) \wedge$
   $\forall (e_1, ..., e_u) \in a : isCompatible(e_1, t_1, TVM) \wedge ... \wedge isCompatible(e_u, t_u, TVM)$
  $\mid a\ BoundedTypeVariable[v, t_1]$ TVM $=$
   $isCompatibleForTypeVariable(a, BoundedTypeVariable[v, t_1], TVM)$

isCompatibleForTypeVariable(a, BoundedTypeVariable[v, t], TVM):
  $if\ v \notin dom(TVM) :$
   $TVM := TVM \cup \{v \rightarrow type(a)\}$
   $return\ True$
  $else :$
   $if\ isCompatible(a, TVM(v), TVM) :$
    $return\ True$
   $else\ if\ let\ lub := leastUpperBound((type(a), TVM(v)) :$
    $TVM := TVM \cup \{v \rightarrow lub\}$
    $return\ True$
   $else :$
    $return\ False$
```

```
leastUpperBound(g, t):
  $\mid case\ object\ \_\ = object$
  $\mid case\ \_\ object\ = object$
  $\mid case\ g_1\ g_2 = getMostSpecific(getSuperTypes(g_1) \cap getSuperTypes(g_2))$
  $\mid otherwise\ Fail$
```

Figure 5.12: Type-checking with type parameters with no bounds

```
def f[$v_1$ <:  $g_1$, ..., $v_n$ <:  $g_n$]($p_1$ :  $t_1$, ...  $p_n$ :  $t_m$) → $t_0$ raises $ex_1$, ..., $ex_n$
```

Figure 5.13: Function type signature supporting type variables

5.6.2 Type checking

We now outline the changes to the type checking mechanism to support type bounds.

To support type bounds with ground types we need to modify the type-checking algorithm presented in Figure 5.12 when a type variable appears as a type for an argument. In such a situation, pytypedecl checks whether the type of the argument is always a subtype of the declared bound for the type variable. This leads to the following updated `isCompatibleForTypeVariable` algorithm presented in Figure 5.14.

```
isCompatibleForTypeVariable(a, BoundedTypeVariable[v, t], TVM):
    $if\ v \notin dom(TVM):$
      $if\ isCompatible(a, t, TVM)$
        $TVM := TVM \cup \{v \rightarrow type(a)\}$
        $return\ True$
      $else:$
        $return\ False$
    $else:$
      $if\ isCompatible(a, TVM(v), TVM):$
        $return\ True$
      $else\ if\ let\ lub := leastUpperBound((type(a), TVM(v)):$
        $TVM := TVM \cup \{v \rightarrow lub\}$
        $return\ True$
      $else:$
        $return\ False$
```

Figure 5.14: Bounded type parameters with ground types

5.6.3 Discussion

Pytypedecl restricts type bounds to be ground types for simplicity. This restriction means typedecl does not support more sophisticated patterns such as allowing a type bound to be another type variable, which can add significant complexity to our system [104]. Our motivation for this restriction is that previous research has shown that advanced use of generics are used little in practice beyond qualified parametric types [101] which were developed in Section 5.4.

5.7 Other features

This section describes other features supported in pytypedecl's type system: overloading, function objects and generators.

5.7.1 Overloading

Pytypedecl allows multiple function type signatures for a single function definition. For example, an `add` operation can be described as follows:

```
def add(a: int, b: int) -> int
def add(a: float, b: int) -> float
def add(a: int, b: float) -> float
def add(a: float, b: float) -> float
```

This description is more precise that using unions which do not express the relationship between the argument types and the return type:

```
def add(a: int or float, b: int or float) -> int or float
```

Pytypedecl uses the multiple signatures to type-check a call instance. It simply checks whether at least one function type signature is compatible. In comparison to other languages like Java, function type signatures in pytypedecl are not used for dispatching. Consequently, there is no need for a selection process or to check for ambiguous function type signatures.

5.7.2 Function objects

Python supports first-class functions. In Python, a function (include a lambda) can be passed or returned from another function as an object. The function object can be distinguished from other objects via its type tag `function`. Pytypedecl uses a special syntax to specify the summary signature of a function object, which is shown in Figure 5.15. It consists of a tuple specifying the argument types in order and the result type. Pytypedecl does not support type variables for function objects's arguments and return type to keep the inference algorithm simple. In fact, a reference to the function object could be used in other parts of the program which would make it difficult to infer a meaningful type for a type variable.

```
t ::= ...  | FunctionType[{$t_1$, ..., $t_w$}, $t_0$]
```

Figure 5.15: Function objects

To type-check the signature of a function passed as argument or returned from a function, pytypedecl creates a new function object that first checks the arguments, then calls the original function, and then checks the return value from that call. This procedure is similar to the algorithm presented in Figure 5.6. This technique is generally referred to as *proxy objects* and previous work has taken a similar approach [57, 108].

5.7.3 Generators

Python supports generators. A generator can be seen as a function that behaves like a lazy iterator. Pytypedecl supports qualified parametric generators, similarly to built-in containers such as `list` and `set`, which we discussed in Section 5.4. To type-check a generator, pytypedecl first creates a proxy generator that calls the generator. This proxy generator checks that the produced value is consistent with the generator's argument type and forwards the value back.

5.8 Implementation

In this section, we explain the implementation of pytypedecl. We start by giving an overview of how to adopt pytypedecl in a Python program. We then outline its architecture.

5.8.1 Overview

Pytypedecl consists of a type declaration language for Python as well as a runtime type-checker. Pytypedecl is available as a standard Python package which can be imported to use the type-checking facilities. The programmer defines a separate *type declaration file* where he provides types declarations for the classes and functions that he wishes to be type-checked. We chose to provide signatures in a separate file instead of mixing them with Python source code for several reasons. First, this approach minimises changes to an existing system by adding new files instead of modifying existing files. Second, it requires a simpler implementation in comparison to parsing a Python file to identify the pytypedecl signatures.

Type declarations	`def MakeAnnouncement(emails:list[str]) -> None`

Python code	`import sys`
	`from pytypedecl import checker`
	
	`def MakeAnnouncement(emails):`
	`    for addr in emails:`
	`        cls.SendEmail(addr)`
	
	`checker.CheckFromFile(sys.modules[__name__], __file__ + "td")`
	
	`if __name__ =="__main__":`
	`    MakeAnnouncement("name@example.com")`

Error with pytypedecl	`TYPE_ERROR:`
	`Function: MakeAnnouncement,`
	`parameter: emails => FOUND: str but EXPECTED: list[str]`

Figure 5.16: Pytypedecl usage example

To use pytypedecl, the programmer needs to follow two steps. First, create a type declaration file with the extension `.pytd` that mirrors the name of the Python file. This type declaration files specifies the signatures of the functions to type-check. Next, the programmer needs to explicitly call the type-checker in the Python file by importing the pytypedecl Python package. The type-checker will then automatically insert runtime checks. Note that pytypedecl lets programmers also use the type declaration file as documentation without making modification to the Python file (e.g. if they decides not to use the type-checking facility).

We illustrate in figure 5.16 how to use pytypedecl. First the programmer specifies the type declarations for the functions he wishes to type-check. In this case, the function `MakeAnnouncement` should take a list of strings as argument. Next, the programmer explicitly calls the type-checking mechanism by importing the pytypedecl package in the appropriate file. In this case, the result is a type error because the function `MakeAnnoucement` was given a string as argument instead of a list of strings. Note that the code would otherwise be accepted and wrongly executed by the Python interpreter.

5.8.2 Architecture

Pytypedecl is implemented as a Python package. As a result, programmers can simply import it just like any other Python package to use it. Our prototype consists of two modules. The first module is responsible for parsing function type signatures in pytypedecl. The second module is responsible for adding checks to the user code using the function type signatures. Figure 5.17 depicts the architecture of pytypedecl.

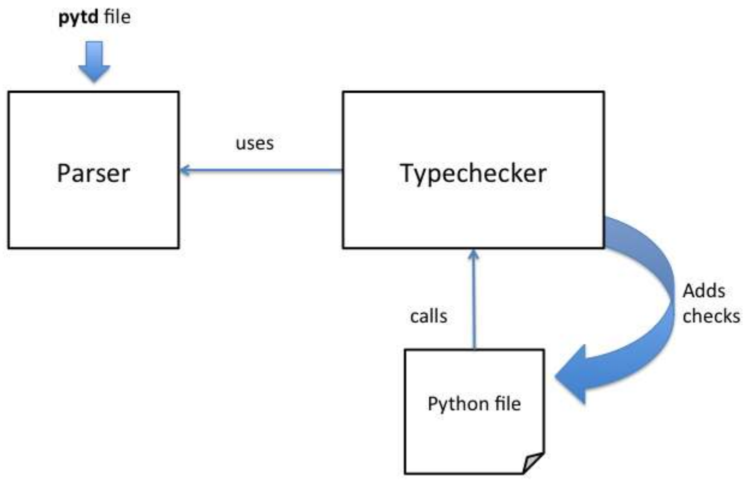


Figure 5.17: Pytypedecl architecture

The parser module is written using a parser generator. We chose to use a parser generator as it is simpler to maintain and make updates to the grammar of the function type signatures. The type-checker module implements the system we outlined in the previous sections. It makes use of Python reflective features to iterate through the functions in a given file. Each function definition slot is replaced with a proxy function object that performs type-checking on the original function.

5.9 Evaluation

This section evaluates pytypedecl on a set of programs and libraries which we annotated with function type signatures. We report the ease of the annotation process, the type annotations used and the performance overhead of pytypedecl. We then discuss limitations of pytypedecl. The next section discusses how pytypedecl can be used for locating incompatibilities in a program for different versions of Python.

5.9.1 Case study targets

To the best of our knowledge, there is no standard benchmark available for conducting studies of code artefacts in Python like Qualities Corpus [120] for Java which we used in the previous chapter. Consequently, we selected three open-source Python program as case studies by performing a search on Github. Our selection of these programs was based on two criteria. First, we selected programs by searching different domains on Github: games, maths library and graphics. The intention was to represent different possible style of programming in Python and explore whether this affects the resulting type annotations. Second, the programs had to be of reasonable size ($\leq$ 5000LoC) so we could manually analyse them. We manually annotated these programs using pytypedecl. The programs have various sizes and represent different uses of Python.

- Sudoku-Solver: a program for solving sudoku puzzles
- Python-colormath: a Python module that provides common colour math operations.
- Latimes-calculate: a Python maths library for journalism

Application	Description	LoC
Sudoku-Solver	Game	399
Latimes-calculate	Numerical library	1287
Python-colormath	Graphics	3448

Figure 5.18: Summary of case study targets

5.9.2 Methodology

We now outline how we conducted the evaluation. As described at the beginning of the chapter, we developed our tool to help developers identify programmer errors. We argued that our system should have two main requirements:

1. Able to express various common use cases for dynamically-typed language
2. Convenient to integrate with existing Python programs

To satisfy these requirements we argued for two design decisions. First, we designed a rich type declaration language with features inspired from other programming languages as discussed in Section 5.2. Second, we argued for a lightweight type-checking system which only monitors the input and output behaviour of functions and is implemented as a library.

To evaluate our system, we executed the program's main entry point or test suite if available. We then explored the source code of the programs and manually annotated them using pytypedecl function signatures. We report the performance impact of using our tool, what features of the type declaration language were used in practice and our experience with the annotation process.

5.9.3 Results

This section describe our findings. Table 5.19 summaries the the evaluation results.

Application	execution time (s)		# annot. fcts.	Features							
	unann.	annot.		ε	$\cup$	$\cap$	ι	[]	$\forall$	<:	Ω
Sudoku-solver	0.58	1.2	18	-	-	-	-	15	-	-	-
unitrest	5.6	5.8	14	-	3	-	-	-	-	-	-
python-colormath	0.25	0.64	22	5	2	-	2	-	-	-	-

Features: ε: Exceptions, $\cup$: Union, $\cap$: Intersection, ι: Interface, []: Qualified parametric types, $\forall$: Parametric polymorphism, <:: Bounded, Ω: Overloading

Figure 5.19: Summary of case studies results

Efficiency The maximum recorded performance overhead of running pytypedecl was 3x the original execution time. Previous work have reported much larger overhead using heavier use of wrappers and smaller programs [108]. This suggests that a lightweight type-checking mechanism based on monitoring of inputs and outputs of function is practical to use.

Type annotation features We found that most type annotations were using features from the basic system presented in Section 5.3. Nonetheless simple parameterisation of containers was common in the sudoku project to model a grid. We also found that declaring an interface `Iterable` in pytypedecl was useful as a particular function in the *Python-colormath* project was checking whether the function argument supported the `__iter__` operation:

```
hasattr(wp_src, '__iter__'):
```

The type annotations used did not include more advanced features such as parametric functions, bounded type variables and overloading. This results support previous research that found that generics and overloading are used less frequently or dependent on the application domain [101, 62].

Annotation process The process of annotating the programs was relatively straightforward. We examined the arguments and results of functions and their use to infer their types. Certain functions in the examined programs had type information as comments which could be re-used and facilitated the annotation process. In certain cases, we had to iterate the annotation process as we missed less obvious cases. For example, the `unirest` program's function expected mostly `str` values and occasionally `unicode` objects. Even though the process was more time consuming for a larger program like `Python-colormath` we expect the original authors of the function to be faster at the annotation process.

We also found that certain parts of code were doing manual type checking, which suggests an automatic system like pytypedecl can be beneficial to developers. For example, the `Python-colormath` program was explicitly checking the arguments of the function inside its body:

```
if isinstance(target_cs, str):
  raise ValueError("target_cs parameter must be a Color object.")
if not issubclass(target_cs, ColorBase):
  raise ValueError("target_cs parameter must be a Color object.")
```

During the annotation process we also found an inconsistency in the names of function arguments with respect to their types. For example, one function accepted a argument `rowNum` which accepts `int` values but another one accepted an argument named `blockNum` which accepts `float` values. Type annotations can be useful to highlight similar discrepancies.

5.9.4 Limitations

In this section, we discuss limitations of the design of pytypedecl and its implementation. We classify the limitations into two groups: coverage of errors and unsupported Python features.

First, our type-checking mechanism is designed to only monitor the input and output of functions. Consequently, our approach is well-suited for programs which are written in a functional-style and makes use of immutability. For these programs, our approach provides a lighter-weight alternative to a traditionally heavier blame-tracking approach and provides similar error-finding capabilities. However, our approach is unable to report errors due to side-effects within function bodies. Nonetheless, as we mentioned in Section 5.3, this can be partially addressed by also validating the types of parameters at the exits of a function as a way to check for possible mutations. Another limitation of pytypedecl is that it is not able to track assignments to attributes of objects. The impact of this limitation can be reduced if accesses to attributes in the Python program were refactored to getter and setter functions, which can be type-checked using pytypedecl.

Second, our implementation does not support certain Python features. Pytypedecl does not support nested functions. For example, in the code below, it is not possible in pytypedecl to type-check the function `addHeader` and `addFooter` because they are defined within the function `format`.

```
def format(s):
  def addHeader(s):
    return "Hi " + s.title()

  def addFooter(s):
    return s + "\nKind regards"

  return addFooter(addHeader(s))
```

A possible solution is to implement monitoring of nested functions using introspection. Nested functions are made available via a tuple of code objects within the parent function. However, this tuple is immutable. Consequently, one would have to create a copy of the parent function with an updated tuple of code objects that contains a wrapped version of the nested function. The implementation of this mechanism results in additional runtime overhead and complexity. Moreover, nested functions are not publicly visible and meant for use as helper functions. Consequently developers can type-check the parent function which makes use of the nested functions as part of its implementation.

In addition, pytypedecl implements a decorator mechanism to enhance functions with type checking by wrapping function definitions with type-checking capabilities. However, programmers can also decorate functions using a built-in Python annotation feature to extend the behaviour of a function. Consequently, our mechanism conflicts with a function that is already explicitly decorated. For example, in our case study we found that the program *Python-colormath* was making use of decorators. The code below uses a user-defined decorator named `color_conversion_function` to return a new function which extends the decorated function with modified attributes:

```
@color_conversion_function(CMYColor, BaseRGBColor)
def CMY_to_RGB(cobj, target_rgb, *args, **kwargs):
  ...
```

Another potential limitation which could prevent adoption of pytypedecl is that of object identity (i.e. reference equality). In Python, object identity can be checked using the `is` operator. The type-checking mechanism we developed in pytypedecl extends functions by creating a new proxy function which extends the original one. As a consequence, the created function has a different identity to the original one, which may break code that relied on object identity. We also explained that features such as generators and attributes can be implementing using a similar technique.

Finally, our implementation only supports positional arguments. Nonetheless, we found in the programs we analysed that named arguments were also occasionally used. Our system could be extended with support for named arguments by incorporating a store mapping parameter names to their associated types as part of a function signatures. A call instance can be type-checked by looking up the type expected for a given parameter name using the store.

5.10 Compatibility checking

In this section, we discuss how a type checking system like pytypedecl can be used by developers to report incompatibilities introduced in different language versions. While we focus on Python 2 and Python 3, the approach is language agnostic. This section starts by providing background on Python 2 and Python 3. It then discusses several use cases of incompatibilities involving different type tags and how pytypedecl can report easier-to-understand error logs.

5.10.1 Python 2 vs. Python 3

The Python community is currently facing a dilemma. There are currently two versions of Python: Python 2 and Python 3. The latter version brings many backward incompatible changes to Python 2 in order to clean up the language. In other words, the same program written in Python 2 is unlikely to run in Python 3. Consequently, upgrading to Python 3 requires an extensive analysis and corrections of the new errors thrown when running a program written for Python using the Python 3 interpreter.

The incompatibilities introduced in Python 3 fit in the ‘behavioural’ and ‘source’ categories that we presented in Chapter 3. Source changes include changing some statements to become built-in functions and exception handling having a different syntax. Behavioural changes include several function returning different object types. The major change however is data related: all text strings are unicode by default. As a consequence of these backward incompatible changes, upgrading a codebase to Python 3 can be quite daunting. There are static tools such as 2to3, which aims to automatically fix certain incompatibilities. However, because of Python’s dynamic typing they are often not good enough to locate all problems because some errors can only be found during program execution. In addition, the errors reported can be cryptic and do not locate the exact source of the problem. The following quotes taken from issues on Github and blogs illustrate this problem.

- on unicode: *"this will be maybe an impossible task to do because of one feature: unicode strings. We currently extensively use unicode [...]."*[27]
- on using 2to3: *"this is something that 2to3 does not pick up and most likely you will not either. At least in my situation it took me a long time to track down the problem because some other implicit conversion was happening at another place."*[23]

5.10.2 Incompatibilities

We now describe the three main semantic incompatibilities in the changes from Python to Python 3. In addition, we exemplify how a type-checking system like pytypedecl can help identify sources of incompatibilities.

Unicode and binary data Strings in Python 2 can be seen as holding either textual data or binary data. In fact, the type `str` and `bytes` are aliased. However, in Python 3 strings are now always unicode and there is a distinction between textual data (`str`) and binary data (`bytes`). In Python 2 there is a separate type `unicode` to distinguish unicode data. This change can lead to confusing code behaviour because `bytes` in Python 2 are a sequence of characters but a sequence of integers in Python 3. In addition, functions such as `open()` can return either `str` or `bytes` in Python 3 depending on the opening mode. However, in Python 2 `str` and `bytes` are the same. Furthermore, mixing textual data and binary data in Python 3 leads to runtime errors (e.g. concatenating a `str` object and a `bytes` object is not allowed). Figure 5.20 illustrate this problem. The code fails because in Python 3 the `open` function returns a binary string. Comparing a unicode string ”hello” with a binary string ”hello” returns `False`, which is a valid comparison in Python 2.

Figure 5.20: str vs. bytes

Type declarations	def assertEquals(s1: str, s2: str) -> bool
	def slurp(filename: str, mode: str) -> str

Valid Python 2	import sys
	import checker # pytypedecl package
	
	def assertEquals(s1, s2):
	  assert(s1 == s2)
	
	def slurp(filename, mode):
	  return open(filename, mode).read()
	
	# opt for typechecking
	checker.CheckFromFile(sys.modules[__name__], __file__ + "td")
	
	data = slurp("data.txt","rb") # "hello"
	assertEquals("hello", data)

Error in Python 3	File "open.py", line 8, in <module>
	    assertEquals("hello", data)
	  File "open.py", line 2, in assertEquals
	    assert(s1 == s2)
	AssertionError

Error with pytypedecl	TYPE_ERROR: Function: slurp, returns <class ’bytes’> but EXPECTED <class ’str’>

Integers In Python 2 an expression such as `1/2` returns `0` (of type int) as both arguments of the division operator are ints. However, `1.0 / 2` returns `0.5` as one of the argument is a `float`. To contrast, in Python 3 the division operator arguments are always converted to `float` and the result is `0.5` in both cases. Figure 5.21 illustrates the problem.

In addition, Python 2 distinguishes between two kinds of integers: `int` (size of 32 or 64bit) and `long` integers (unlimited size). In Python 3, both kinds are unified into the type `int` which behaves like Python 2's long (except that the printable version using the function `repr()` no longer has a trailing `L`). The type `long` disappears from Python 3.

Figure 5.21: Integer division

Type declarations	`def pivot_index(len: int) -> int`

Valid Python 2	`def pivot_index(len):`
	`  return len/2`
	
	`list = [4, 1, 3, 10, 4]`
	`index = pivot_index(len(list))`
	`print(list[index])`

Error in Python 3	`  File "integer.py", line 6, in <module>`
	`    print(list[index])`
	`TypeError: list indices must be integers,`
	` not float`

Error with pytypedecl	`TYPE_ERROR: Function: pivot_index, returns <class 'float'>`
	`but EXPECTED <class 'int'>`

Views and iterators instead of list Many built-in functions such as `map`, `filter` and `zip` return a `list` in Python 2. To contrast, in Python 3 they return an `iterator` which is more memory efficient. However, iterators have different semantics and functions that can operate on them. In addition, iterators are lazy and can be consumed only once in comparison to a list that is eager and can be consumed multiple times. The Python 2 semantics can be preserved in Python 3 programs by manually converting the result of these functions from an `iterator` to a `list`). Nonetheless, the preferred idiom in Python 3 is to use iterators instead of the less memory efficient `list` when possible. Furthermore, in Python 2 the methods `keys()`, `values()` and `items()` on a dictionary return a copy of the elements inside a `list`. In Python 3, they now return a `view` which can access the original elements. Figure 5.22 illustrates the problem.

Figure 5.22: Dictionary and views

Type declarations	`def concat(l1: list, l2: list) -> list`

Valid Python 2	`def concat(l1, l2):`
	`  return l1 + l2`
	
	`d = {1:'a', 2:'b'}`
	`l = concat(d.keys(), d.values())`

Error in Python 3	`  File "dict.py", line 5, in <module>`
	`    l = concat(d.keys(), d.values())`
	`  File "dict.py", line 2, in concat`
	`    return l1 + l2`
	`TypeError: unsupported operand type(s) for +:`
	`'dict_keys' and 'dict_values'`

Error with pytypedecl	`TYPE_ERROR: Function: concat,`
	`parameter:`
	`l1 => FOUND <class 'dict_keys'> but EXPECTED <class 'list'>`
	`parameter:`
	`l2 => FOUND <class 'dict_values'> but EXPECTED <class 'list'>`

5.11 Related work

Many type systems have been proposed in the literature to enhance dynamically-typed languages with additional type-checking. This section reviews existing tools that are related to our work.

Reticulated Python brings gradual typing to Python [135]. It is a source-to-source translator that accepts Python 3 code. It uses a combination of type annotations on functions and type inference for local variables to generate code which adds type-checking to the original program. It provides three different dynamic type casts to deal with parts of code that can not be statically checked. The first mechanism is called *guarded semantics* which wraps a value into a proxy object. The proxy object contains information such as line number and error message that will be reported if type-checking fails. The second mechanism is called *transient semantics*, which checks the type of a value without a proxy object. The third mechanism, is named *monotonic semantics*, which is similar to the guarded mechanism but instead of using proxy objects, type information is stored with the original values themselves as a tag. In comparison to our work, pytypedecl uses a combination of transient semantics to check the input and output of function definitions and also proxy objects to implement typechecking for generators and function objects. In addition, Reticulated Python's type system does not support generics.

The Ruby Type Checker (*rtc*) is a tool that type-checks ruby programs at runtime using explicit type annotations [108]. Rtc also checks types at function entrances and exits. However, rtc uses proxy objects. In other words, each type-annotated object is wrapped by a proxy object which performs typechecking. In comparison, pytypedecl uses a simpler approach where values are type-checked immediately at function boundaries. Nonetheless, generators and function objects are also implemented using function objects.

In addition, pytypedecl's type-system also supports interfaces and noneable types as we described in Section 5.3. Finally, types in pytypedecl are declared in an external file. This approach enables programmer to gradually annotate their code without committing changes to code files.

Mypy is a language that has a Python-compatible syntax and implements several Python's features [78]. In comparison to our work, mypy supports type inference and explicit type annotation to statically check programs. However, the motivation behind pytypedecl and mypy is different. Pytypedecl was designed as a library so that it integrates with an existing toolchain whereas mypy runs as a separate program. Pytypedecl and mypy share similar type declaration features such as generics, overloading and union types. However, mypy does not support bounded type parameters and interfaces.

Cython is a superset of Python that is focused on performance [36]. Programmers can annotate Cython code with explicit types, which are then compiled to C. However, Cython introduces several incompatibilities with Python syntax. In other words, a Cython program may not run with a Python interpreter. In comparison to our work, Cython's type system features are different. For example, it does not support exceptions, generics or interfaces. In addition, Cython's type syntax was designed to feel familiar to C programmers. Pytypedecl's type declaration language was designed to feel "Pythonic".

Julia is a dynamic language which supports explicit type annotations [39]. Pytypedecl's type system has similar features to Julia's type system but also fundamental differences. For example, both Julia and pytypedecl support optionally annotated types, union types, generic functions and bounded type parameters. However, Julia and pytypedecl deal with ad-hoc polymorphism differently. Julia has an algorithm for selecting the most-specific method from a list of overloaded signatures to resolve which method body to execute. In pytypedecl we similarly allow multiple signatures but only have one method body. In addition, pytypedecl treats parametric types to be covariant by default and supports other type system features such as intersection types, noneable types and interfaces.

5.12 Conclusion

This chapter described a system called pytypedecl which uses type annotations that are checked during the program's execution and reports errors. Our system has two main characteristics. Firstly, it consists of a rich type declaration language to express function type signatures for function definitions in a program. Secondly, type-checking happens at function boundaries. We implemented the type-checking mechanism as a lightweight library to minimise the overhead of integrating our system into existing projects. We showed how the use of type annotations as contracts can be useful to locate errors appearing when upgrading to a new language version. We illustrated this with supporting examples in Python 2 and Python 3. Finally, the work presented in this chapter influenced the design of a standard type annotation language as part of a Python 3 enhancement proposal by Rossum et al [132].

Chapter 6

Dynamic discovery of function type signatures for Python

In the previous chapter, we presented a runtime technique for Python which dynamically type checks function calls against statically declared signatures. The type checking mechanism described used a list of function type signatures that act as contracts for the input and output of the functions defined in a program. We showed that this technique can support developers in locating the source of potential errors when running a problematic program. Such errors include programming errors as well as incompatibilities introduced when migrating to a different language version. In addition, we argued that the set of function signature declarations is also useful for documentation and future maintenance of the program.

So far, the function type signatures were manually provided by the programmer. For new projects, writing signatures alongside function bodies can naturally be part of the development process. Nevertheless, the manual process of adding function signatures can hinder the adoption of this technique in existing large projects. In fact, manually annotating functions with type signatures requires a good understanding of the internals of the project and can be time-consuming. In addition, large projects are often developed by multiple programmers at different periods of the project lifespan. Worse, programmers who were involved with the project may not be available anymore to document it.

This chapter contributes an automated approach for generating function type signatures to alleviate the burden for programmers. We present a lightweight mechanism which monitors the input and output behaviours of functions as the program is run. This information is then amalgamated to produce function type signatures. We implemented and evaluated this technique on several Python projects. We found that this technique provides sufficient insights about the program in order to reduce manual annotation. In addition, the generated output is manageable by programmers. This technique is also language-agnostic and can be implemented in other programming languages.

6.1 Introduction

Type checking is useful to locate the source of errors in a program. Typically, a static type checking system relies on explicit type annotations for verification. In addition, type annotations have shown to be useful for documentation and maintenance of programs. These type annotations are usually provided manually by the programmer. However, several systems also provide type inference where type annotations can be deduced automatically. As a result, type inference lightens the burden on programmers to provide type annotations themselves.

In the context of dynamically-typed languages, there have been several attempts at bringing type inference to languages like Python. Traditionally, this research has focused on implementing type inference as a static analysis [60, 114]. In other words, type information about the program is deduced without actually executing the program.

Nonetheless, static type inference for dynamically-typed languages can also have several drawbacks for programmers. First, most dynamically-typed languages have subtle semantics and features which make it difficult to develop static type inference. For example, it is possible for a variable's type to differ along different paths such as inside an `if` block:

```
function process(id):
  if(condition):
    return "2"
  return 1
```

In addition, it is possible for programs to add and remove attributes at runtime. Moreover, programmers my use other reflective features such as `eval` to evaluate an expression at runtime and `exec` to evaluate a text string as code to execute. For example, the following program will evaluate an expression entered by the user (e.g. "1 + 2") using `eval`. It will then execute the code passed as a string to `exec`.

```
e = eval("input()")
a = eval("1 + 1 + e")
exec("b = a * 2")
print b
```

As a result, implementations of static inference systems often only address a subset of the language and provide approximative results, which can hinder the benefits of its output in terms of documentation and type checking. More sophisticated systems tend to perform slower because they need to run more complex analysis.

Another downside of static type inference systems for dynamically-typed languages is that they are generally implemented as a separate tool or even a different language with a separate compiler. Consequently, they are difficult to integrate with a company's existing build system and internal policies which can prevent the adoption of these tools.

To address these issues, this chapter argues for a lightweight *dynamic* approach which infers function type signatures by monitoring the execution of the program. We present a system which complements the work presented in the previous chapter. In fact, the system we present also tracks the input and output behaviours of the functions when the program is executed. However, instead of using this information for verification, it is amalgamated to produce function type signatures for the functions defined in the program. Figure 6.1 outlines the process in simple terms.

We show how this technique can be implemented as a library which facilitates integration in an existing tool chain. We found that monitoring the input and output behaviours of functions can result in a lot of information to manage for programmers. For example, we evaluated our tool with a program of over 30,000 lines of code, which resulted to an average number of 2.5 function type signatures per function definition. To address this

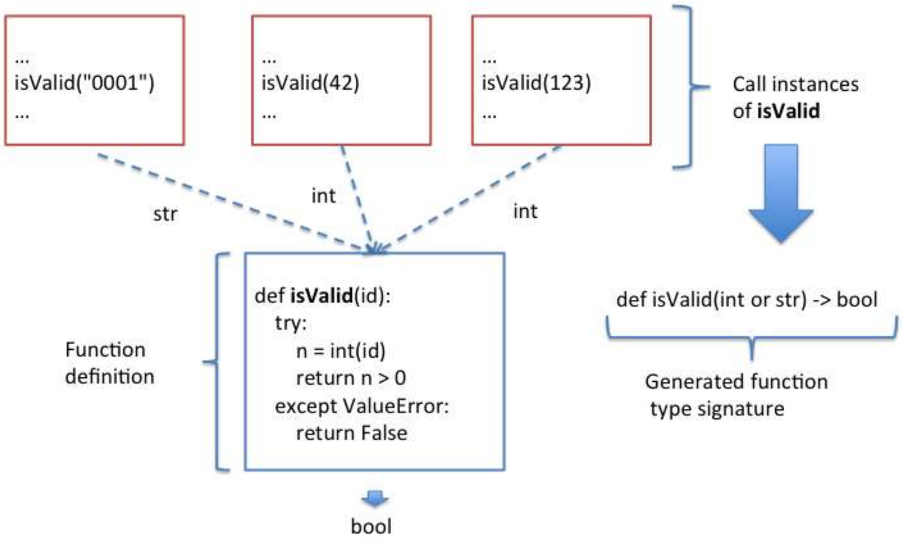


Figure 6.1: Monitoring of functions

particular issue, we describe several techniques which amalgamate the collected type information to reduce the number of function type signatures presented to programmers. Our evaluation show that these techniques can significantly reduce the number of function type signatures per function as well as still produce enough useful insights for programmers. In addition, we found that the performance overhead of our system was manageable.

The main contributions of this chapter are as follows:

- The presentation of a lightweight dynamic system for monitoring and amalgamating type information of dynamically-typed languages
- A set of techniques for collapsing a list of call instances into manageable function type signatures
- An implementation and evaluation of our system for Python

The rest of this chapter is structured as follows. Section 6.2 presents the architecture of our monitoring system. Section 6.3 tackles the question of what information needs to be collected. Section 6.4 tackles the questions of how to collect the required information. Section 6.5 discusses how this information is amalgamated to produce manageable function type signatures. Section 6.6 describe the implementation of this system for Python. Section 6.7 evaluates our system for Python programs from different application areas and sizes of up to 25,000 LOC. Finally, Section 6.9 describes related work.

6.2 Architecture

This section outlines the architecture of our type monitoring and amalgamation system called *pytypemonitor*. Its purpose is to infer function type signatures. The design of the system is programming language agnostic, however, this chapter focuses on Python.

We divide the system into three distinct modules. Each module is decoupled from one another in a similar manner to Daikon's architecture [55]. As a result, each module can be improved on separately, extended or replaced with a different implementation. Figure 6.2 presents a diagram of the architecture.

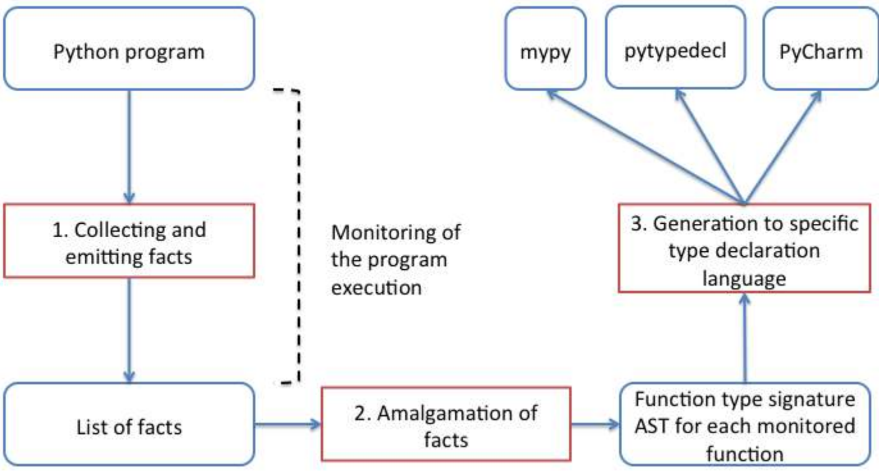

Python program
1. Collecting and emitting facts
List of facts
Monitoring of the program execution
2. Amalgamation of facts
Function type signature AST for each monitored function
3. Generation to specific type declaration language
mypy
pytypedecl
PyCharm

Figure 6.2: Architecture

Collecting and emitting facts This first module is responsible for collecting source-code information about the Python program. We will refer to this information as *facts*. There are several possible techniques to collect the facts including instrumenting the program, static analysis or runtime monitoring. We outline them in the next section. This module generates a list of facts which can becomes the input of the second module. We discuss what facts to collect in order to provide useful insights about the program in Section 6.3 .

Amalgamation of the facts The second module is responsible for processing and summarising the facts collected by the first module. For example, it may apply various transformations such as removing duplicates or combining facts together. In other words, the second module is analogous to a compiler's optimisation phases. It then presents the result to a third module that is responsible for the textual generation of the function type signatures.

Generation The third module reads the result from the previous module and generates function signatures in a specific format. We presented a type declaration in the previous chapter, however, one has the flexibility to use a different type declaration language if required. For example, in the previous chapter we discussed other type declaration proposals for Python including mypy [78] and PyCharm [24].

6.3 Required facts

Before we outline the mechanisms to collect information from a Python program, we need to determine what information is required to produce useful insights about the program. We refer to this information as *source-code facts* and describe them in this section. In the next section, we present our mechanism to collect the facts.

Our system monitors the input and output behaviour of functions. When a function is called in the program, we refer to this as a *call instance*. There are four essential facts about a given call instance that we can collect:

1. type of the receiver (i.e. the type of the object the function or method is called on)

2. types of arguments at entrance and exit

3. return type

4. exceptions raised

Note that a particular call instance can only produce a 3-tuple of these facts because it either returns a result or throw an exception.

To track the input behaviour of a function we record the types of the arguments of the call instances. As we explained in the previous chapter, it can also be useful to keep track of whether the types of arguments are changing during the execution of a function.

In addition, to produce a function type signature we need to locate the function definition associated with a call instance. For this we need to record the file and class where that function or method is defined.

We presented the type declaration language of pytypedecl in the previous chapter to describe function type signatures. We now detail what further facts are useful to collect to express function type signatures that make use of the various features in the type declaration language such as generics, union types and overloading.

Containers The type declaration language described in the previous chapter includes both heterogenous containers and homogenous containers. Heterogenous containers can have elements of a variety of types. For example, the types `list` or `dict` describe heterogenous containers. On the other hand, homogeneous containers have elements of a single type or subtype of a common type that is different than `object`. For example, the type `list[int]` describes a homogeneous list that contains only elements of type `int`. As a result, to differentiate between the two kinds of containers we need to track the type of each element for all containers coming in and out of a function.

Objects The type of each argument can be determined using the built-in `type()` function in Python. If the type returned is a non-builtin type (e.g. of a user-defined class), we can record the name of the instance's class by inspecting the attribute `__class__` of the object. There are two additional facts that are useful to collect to summarise call instances into function type signatures.

First, we can collect the type hierarchy for each new type discovered at the time of call. The type hierarchy can be useful to avoid a union for each call instances when they share a common supertype. We explain in Section 6.5 how this transformation is implemented. As an example, if a function `eat` is called with an object of type `Apple` and another object of type `Banana`, we can realise that both `Apple` and `Banana` share a common supertype `Fruit` and summarise the two call instances below into a single function type signature for `eat` which only takes a `Fruit` object as input:

```
def eat(f):
  pass

eat(Apple())
eat(Banana())
```

Second, if the types do not have an explicit nominal subtyping relation, they may still have an implicit structural subtyping relation if their sets of operations intersect. Our type declaration language supports this idea via interfaces. As a result, it is also useful to record the operations supported by the objects coming in and out of a function. Note that Python allows new attributes to be added to objects at runtime, which may make it more difficult to infer a structural subtyping relation. We can mitigate this by tracking the operations both at entrance and exit of a call instance to gather further insights.

Other useful facts There are other interesting facts that can be collected using more complex monitoring. We outline them briefly. However, pytypemonitor does not support them to keep a low monitoring overhead.

- *Function objects*: Python supports first-class functions in two forms. First, one can pass an existing named function as an argument to another function. Second, one can also pass a lambda expression. In both cases, the argument will have the Python type tag `function`. However, function objects can themselves take arguments, return a result and throw exceptions. To track this information, we would need to keep track of the arguments and output of all function objects passed as an argument or returned from a function.

- *Generator*: Generator objects behave like iterators and produce values. They can be seen as functions which return a new value at each call. One could monitor the type of each value produced by a generator to infer whether it is a homogenous generator or heterogenous generator.

- *Attributes*: It is possible to also monitor the attributes (i.e. properties) of each object coming in and out of a call instance. This provides further insights about the structure of the data types available in the program.

6.4 Collecting source-code facts

We now outline how we monitor the input and output behaviour of call instances to collect source-code facts. We wrap each function definition within a proxy. This section starts by reviewing alternative monitoring mechanisms and then presents our approach. Next, we describe how the facts are represented so they can be processed by the amalgamation module to produce function type signatures.

6.4.1 Collection mechanisms

There are several possible mechanisms to collect information about the program. We review the pros and cons of each mechanism in relation to our goal and then present the approach we developed.

Static analysis The main advantage of static analysis is that it does not require to run the program to infer type declarations. As a result, static analysis does not impact the runtime performance of the program. However, there are a few downsides. First, the programmer needs to run an external tool to run the analysis, which may be inconvenient to integrate with an existing build system or continuous integration environment. Second, static analysis typically provides over-approximations as it is not based on actual program runs.

Instrumenting Another approach, called instrumentation, is to place several hooks inside a program to monitor required information. Typically this technique is implemented by a program transformation step which rewrites the original program and adds code to monitor program points of interest. Adding instrumentation to a program at compile-time is fast in comparison to a whole program analysis. The downside of this approach is that it requires pre-processing the existing source code using an external tool. This approach was selected by Daikon, JSTrace and Mino [55, 123, 113] which we review in Section 6.9. In addition, it typically provides incomplete result as it is dependent on the the code coverage. Instrumentation also adds a runtime overhead to monitor information.

Tracing Several dynamically-typed languages come with a built-in module which allow code tracing. It is a mechanism which provides information about the execution of the application at runtime. For example, Python includes the module `sys` which allows to place various program hooks at runtime and inspect stack frames of functions being called. It is typically used to implement debugging facilities. Unlike instrumentation, it does not require to transform source code to add hooks inside the program to monitor.

Proxy objects Another technique which we developed in the previous chapter is to dynamically wrap interesting program elements via *proxy objects* at runtime. In other words, at runtime, each function definition is decorated with monitoring capability. Consequently, call instances of that function are tracked and can also dispatch to the original function. This is the approach we selected as a lightweight mechanism to monitor the input and output behaviour of functions. It can be simply implemented in Python thanks to its reflection capabilities. The advantage of this approach is that it can be implemented as a library which is simple to integrate with an existing development infrastructure. However, it brings additional runtime performance overhead in comparison to a purely static analysis approach. Note that this technique has lower overhead compared to full dynamic taint analysis. Indeed, taint analysis requires to propagate the information along the program's execution. We detail the implementation of our mechanism in the next subsection.

6.4.2 Representation of the set of facts

We now discuss a suitable format for representing and storing facts about the program as discussed in the previous section. We outline two criteria that the representation should meet in terms of structure and extensibility. First, the representation should allow easy traversal of the data. In other words, the representation should facilitate lookups of information. Second, the representation should be extensible for additional new facts that may need to be recorded.

Given these two criteria, we use a representation which is a mix of a *graph* for hierarchical information and a *document-oriented* structure.

In relation to our work, in chapter 4, we argued that a graph representation was suitable for querying and storing hierarchical information. We showed that type hierarchy can be simply queried with a language that supports transitive closure. This is why we propose a similar graph representation for the purpose of recording the type hierarchy in a Python program. However, unlike the type hierarchy, information about a call instance does not exhibit deeply connected structures. This is why we propose and use in our prototype a document-oriented approach based on a JSON structure constructed from key-value pairs for this information. In the JSON format, a key is a text string representing a named property and a value can be either a text string, an ordered list of values or a document itself.

In addition, a document allows for semi-structured data. In other words, documents may share structural similarity but do not necessarily adhere to the same schema. This property allows us to extend a particular document on an ad-hoc basis with any additional information that may be useful.

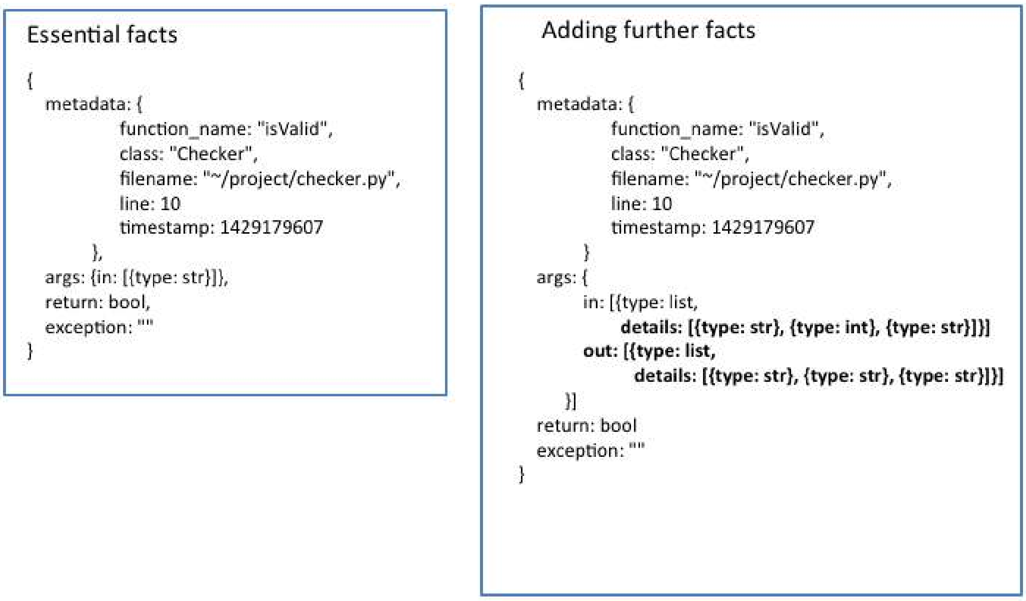
Data collected at a call instance has three main components:

1. Metadata about the function including its name, its associated class if any, the program location of the call instance and timestamp

2. Parameters of the function including their positions and the types at entry

3. The return type of the function or any exception that was raised

The first component can be simply modelled as a set of key-value pairs. The second component can be modelled as a list that stores the types of the arguments sorted by the argument index position. Each element in that list is an entry recording the type of the argument and the index of the element represents the position of the argument in the call instance. Finally, the the third element can be also simply modelled as a key-value pair. The fact that a document is semi-structured data allows us to extend the document in the future with any additional information that may be useful. This is an essential characteristic for our system because it is designed for agile modification. For example, if we later decide to record additional facts about the arguments of a call instances, we can extend the document to include this information (e.g. the types at exit or the type of each element inside a container to differentiate between a homogenous and heterogenous container). Figure 6.3 illustrates the structure for representing the essentials facts and then extended with additional facts about the argument types.



Essential facts

```
{
  metadata: {
           function_name: "isValid",
           class: "Checker",
           filename: "~/project/checker.py",
           line: 10
           timestamp: 1429179607
        },
  args: {in: [{type: str}]},
  return: bool,
  exception: ""
}
```

Adding further facts

```
{
  metadata: {
           function_name: "isValid",
           class: "Checker",
           filename: "~/project/checker.py",
           line: 10
           timestamp: 1429179607
        }
  args: {
      in: [{type: list,
         details: [{type: str}, {type: int}, {type: str}]}]
      out: [{type: list,
           details: [{type: str}, {type: str}, {type: str}]}]
    }]
  return: bool
  exception: ""
}
```

Figure 6.3: Data representation of a call instance

6.5 Amalgamation of the facts

The described implementation of the facts collection records a possible function type signature for each call instance. This is overwhelming for programmers to manage and get useful insights from. For example, a call-site inside a loop may produce numerous call instances which are producing separate function type signatures. However, in practice several call instances may share the same information such as the same return type and types of the arguments. This section presents five techniques for reducing the output of the monitoring by combining the facts of related call sites. Our evaluation on several programs shows that these techniques can significantly reduce the amount of generated function type signatures. We found that these techniques generated an average number of 2.5 function type signatures per function definition.

6.5.1 Overview

This section briefly describes five techniques for producing useful function type signatures for programmers from a set of call instances encountered during the execution of a program.

The definition of *useful* is difficult to define. In the context of this work, we are looking to produce function type signatures that will serve as documentation for developers. As a result, we are interested in signatures that are maintainable and readable by developers. This is why we prefer conciseness over verbosity, even though readability is also a subjective matter. This means that when there are multiple representation of the same concept, we favour the concise alternative features of the type declaration language when possible. For example, we make use of union types where possible instead of multiple similar overloaded signatures of a function to reduce the noise of the function type signatures produced.

The techniques described can be seen as transformations similar to optimisation phases produced by compilers. As a consequence, the order of the transformation phases can produce different results. We discuss this issue in Section 6.5.7 and evaluate the result of applying these techniques in Section 6.7.

6.5.2 Remove duplicates

The most straightforward transformation phase is to remove duplicate function type signatures. Since a function type signature is recorded for every call instance during the program execution, it is likely that redundant information is produced. For example, given two identical function type signatures:

```
def foo(a: int) -> int
def foo(a: int) -> int
```

We can simply transform them to one function type signature

```
def foo(a: int) -> int
```

Note that the keyword `def` is not formally part of the signature but is convenient for reading.

6.5.3 Disjoint union of results and exceptions types

In certain cases, function signatures produced may have identically list of arguments but only differ by the return type. In which case we can produce a single function signature which takes the union of the two result types. For example:

```
def foo(a: int) -> int
def foo(a: int) -> float
```

can be transformed to the following more compact signature:

```
def foo(a: int) -> int or float
```

We also unify result types and exceptions. Given that a call instance can return either a value or raise an exception, we combine them using a Java-like signature:

```
def foo(a: int) -> int raises Exception
def foo(a: int) -> int
```

becomes

```
def foo(a: int) -> int raises Exception
```

A similar transformation can be applied when several function type signatures raise exceptions. For example:

```
def foo(a: int) -> int raises A
def foo(a: int) -> int raises B
```

can be transformed to the following more compact signature:

```
def foo(a: int) -> int raises A, B
```

Note that the following signatures

```
def foo(a: int) -> int raises A
def foo(a: int) -> float raises B
```

could be transformed to the following compact signature:

```
def foo(a: int) -> int or float raises A, B
```

However, this new function type signature loses precision over the above overloaded versions. With this signature, a call instance that returns `float` and raises an exception `A` will be valid. Note that, the two above overloaded function type signatures for `foo` do not accept this case.

6.5.4 Unify functions by common super classes

Assuming we have a union of several types: $(t_1 \cup ... \cup t_n)$ (through overloading or explicit union type). The type could be simplified to simply `t` if the set $(t_1, ..., t_n)$ represents all the direct subtypes of `t`.

As a concrete example, the following overloaded function type signatures:

```
def enjoy(f: Apple) -> boolean
def enjoy(f: Banana) -> boolean
```

can be first transformed to a union type as follows:

```
def enjoy(f: Apple or Banana) -> boolean
```

Provided the class inheritance graph shows that that the type `Fruit` is a direct supertype and that `Apple` and `Banana` are the only subtypes of `Fruit`, then the signature can be transformed to the following more compact form:

```
def enjoy(f: Fruit) -> boolean
```

However, this transformation raises interesting questions:

- What if there is more than one suitable supertype?
- What if only a partial list of subtypes of `Fruit` occur?

Ambiguous super type If more than one supertype is acceptable, we chose to pick an arbitrary type from the list of acceptable supertypes. However, there are several possible other ways to decide which one to select:

1. Ask the programmer explicitly which type is most suitable.

2. Provide a union of all possible suitable types. This could result in a further optimisation if this set also shares a common supertype that is not `object`.

3. Have a custom rule-based system to make the decision. For example, from a list of possible supertypes, we may want to select the type that appears most frequently in other function type signatures.

Partial list of subtypes It is possible that the analysis only finds a partial list of subtypes. For example, the set $(t_1 \cup ... \cup t_{n-1})$ contains all subtypes of the type `t` except t_n. Can we simplify the type to just `t`? We highlight two strategies to answer this question. In our work, we take an *approximate* approach.

- *conservative*: if only a partial list of subtypes is available then it cannot be replaced by a supertype. In fact, a supertype may accept subtypes which were not supposed to be accepted.

- *approximate*: if a partial list of subtypes is available we may want to replace it by a supertype. There are different levels of confidence that can be used to make the decision. For example, if more than half of the subtypes are available then the programmer may want to replace the union with an appropriate supertype. In practice, if a partial list of subtypes is present it is likely that the supertype is intended. In our work, we selected a heuristic to simplify to the common supertype if at least four subtypes are provided.

6.5.5 Factorise cartesian product of functions

A set of signatures may be a cartesian product expansion of unions. In general, given a list of overloaded signatures and set of types for each parameter position $P_1, ...P_n$, we can transform the set of all signatures $\{(p_1, ..., p_n) \mid p_1 \in P_1 \wedge ... \wedge p_n \in P_n\}$ to the single signature $(P_1 ... P_n)$. Section 6.7.3 shows an example as part of our evaluation.

For example, the following overloaded function type signatures:

```
def f(x: int, y: int) -> bool
def f(x: int, y: float) -> bool
def f(x: float, y: int) -> bool
def f(x: float, y: float) -> bool
```

can be simplified to the more compact form:

```
def f(x: int or float, y: int or float) -> bool
```

6.5.6 Infer parametric polymorphic function

Parametric polymorphic functions are useful both conceptually for structure and for documentation. In fact, they convey that the intent of the function is to abstract over a type. More explicit intent helps improve the maintainability of code. In addition, parametric polymorphic functions are more concise in comparison to equivalent overloaded signatures. For example, given a list of function signatures as follows where the observed argument type and return type coincide to be the same:

```
def identity(x: int) -> int
def identity(x: float) -> float
def identity(x : bool) -> bool
```

We can transform this list of signatures to a more concise and meaningful parametric function signature as follows, where `T` is a type parameter for the function `identity`:

```
def identity[T](x: T) -> T
```

This transformation example raises the following question: given a list of signatures, how and when can we transform them to a parametric function signature? Since our system does not include data-flow information, we use a statistical inference. In our work, we decide to apply a unification algorithm and then have a simple heuristic based on the observed number of identical parametric function type signatures. However, more advanced forms of inference are possible. For example, Message et al. developed a system that uses statistical techniques to make type judgements [83].

First, we distinguish between two levels of confidence based on the information available:

- **conservative**: this mechanism guarantees no false positives. In other words, a function is inferred to be parametric only if the available information demonstrates this.
- **optimistic**: this mechanism allows for false positives. In other words, the available information does not completely guarantee that the function is in fact parametric.

We now describe our system based on a substitution algorithm to infer parametric function type signatures. The algorithm operates in three phases:

1. **Unification**: for each given function type signature, associate a fresh type variable for each distinct ground type (e.g. `int`, `list`, `bool`) that appears more than once in the function type signatures
2. **Factorisation**: combine signatures that have overlapping type variables in similar parameter positions and return type. Any remaining type variables is reverted back to their associated ground type.

In the first step, we assign a type variable from an ordered list of fresh variables ranged over by `A`, `B`, `C` ... `Z` to each new ground type that appears in a parameter as well as in the return position. We apply this process for each function type signature and using the same ordered list of fresh variables. For example, the following signature:

```
def foo(a: int, b: int, c: str) -> int
def foo(a: float, b: float, c: bool) -> float
def foo(a: str, b: str, c: bool) -> str
```

will be transformed to:

```
def foo[A](a: A, b: A, c: str) -> A
def foo[A](a: A, b: A, c: bool) -> A
def foo[A](a: A, b: A, c: bool) -> A
```

This step is effectively a unification system where ground types form atomic terms, each type of a parameter is assigned a variable and the equivalence relation is simply whether the value of the variable are equal.

Finally, the step phase combines identical signatures, which produces:

```
def foo[A](a: A, b: A, c: str) -> A
def foo[A](a: A, b: A, c: bool) -> A
```

Note that the phase *remove duplicates* could also do this final transformation.

This signature can be further compacted by applying the disjoint union phase:

```
def foo[A](a: A, b: A, c: str or bool) -> A
```

6.5.7 Discussion

We described five different transformation phases which aimed to produce more compact function signatures from a collection of recorded facts about the program. However, we did not discuss the order for these phases.

This problem is related to compiler optimisation phases ordering [133]. Different optimisation phases in the compiler can interact with each other. As a result, the choice of order may increase or reduce opportunities for optimisations. In addition, different orders may produce different output code with different characteristics such as speed and memory consumption. For example, should we apply a common-sub expression (CSE) phase before the register allocation phase? The CSE phase may increase the number of variables required and cause a cheap-to-recompute expression to be stored in memory rather than into a register. However, in certain cases CSE may reduce the number of registers required as well. Different techniques have been suggested to find solutions to the phase-order problem including using genetic algorithms [46] and machine learning [76].

Whitefield et al. described a general framework for approaching code-improving transformations [136]. They present a specification language (Gospel) that can express conditions and dependencies amongst transformations to analyse how transformation phases interact.

In relation to our work, different transformation phases orders may produce function type signatures that have different levels of conciseness and readability. We argued earlier that readability is often a matter of taste and is therefore difficult to quantify.

We have therefore chosen an ordering of the transformations phases as defined in Figure 6.4.

Figure 6.4: Transformation phase orders

Order	**Phase**
1	Remove duplicates
2	Disjoint union of results and exceptions types
3	Factorise cartesian product of functions
4	Unify functions by common super classes
5	Infer parametric polymorphic function

We now justify the chosen order.

1. We can always apply the *remove duplicates phase* first as redundant information does not contribute to subsequent phases

2. After that, we factorise function type signatures that have the same argument list but differ in result type and exception raised.

3. Next, we factorise cartesian product of functions. This phase can collapse several signatures into a single form that is more concise.

4. The previous two phases generated several unions which may share a common super class. This is why, we choose to follow up with inferring functions by common super classes.

5. Finally, the resulting signatures can be further collapsed by inferring parametric types

Note that, certain phases could be applied again to further compact generated function type signatures. For example, phase 5 may produce two function type signatures as follows:

```
def foo[A](a: A, b: A, c: str) -> A
def foo[A](a: A, b: A, c: bool) -> A
```

This signature can be further compacted by applying the disjoint union phase:

```
def foo[A](a: A, b: A, c: str or bool) -> A
```

6.6 Implementation

In this section, we give an overview of the implementation of our type monitoring system.

Our prototype consists of two modules. The first module is responsible for monitoring and recording type information of Python programs. The second module is responsible for combining the type information to produce function type signatures.

Monitoring module The first module is implemented as a Python module. It contains one main entry point function called `MonitorFunctions` which iterates through each function and method definitions inside a module and decorates them with monitoring capabilities. Developers need to add the following line inside a Python file to monitor function calls (the Python expression `sys.modules[__name__]` returns pairs of function names and bodies in the current module).

```
monitoring.MonitorFunctions(sys.modules[__name__])
```

The implementation of `MonitorFunction` is summarised as follows where `Functions`, `Classes`, `MonitorFunction` and `MonitorMethod` are helper functions we defined:

```
def MonitorFunctions(module):
    for f_name, f_def in Functions(module):
        module.__dict__[f_name] = MonitorFunction(module, f_def)

    for c_name, c_def in Classes(module):
        for m_name, m_def in MethodsForClass(c_def):
                setattr(c_def, m_name, MonitorMethod(module, c_def, m_def))
```

The helper functions make use of Python introspection features to inspect information about a module. Function definitions in a module can be looked up by iterating the module's built-in dictionary where keys are names of functions and values are objects representing instances of a function definition. Similarly, we inspect all classes and methods of a module using the built-in `inspect` Python module which provides facilities to dynamically examine the content of modules and classes. We then use Python's built-in decorators features to wrap each function and method with monitoring capabilities. Figure 6.5 illustrates the process.

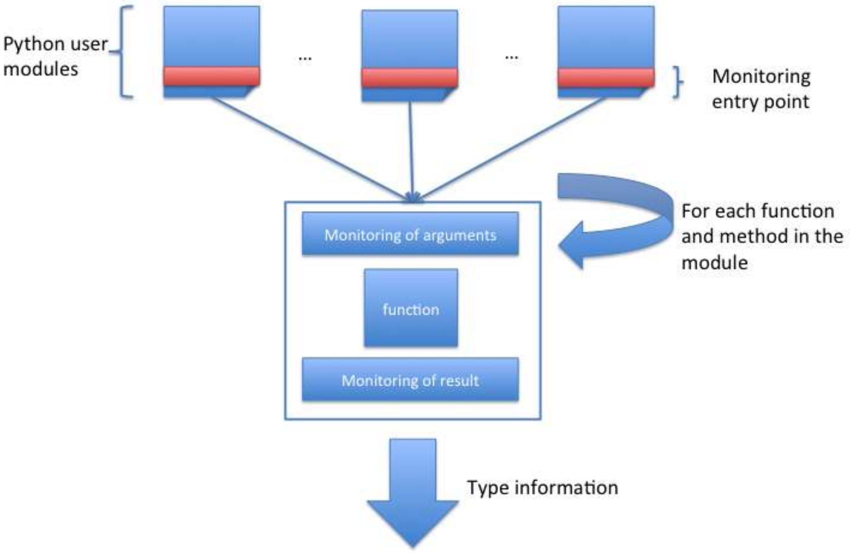


Figure 6.5: Monitoring of python modules

Almagamation module We implemented the amalgamation module in Java. Each reduction phase is implemented as a function that takes as input a list of function type signatures and produces as output another list of function type signatures. This enables the user to chain phases in a customised ordering. The implementation of the reduction phases make use of Java 8's Stream API which provides many common data manipulation out of the box such as filtering, duplicate removal and grouping.

6.7 Evaluation

In this section, we evaluate our system. We performed case studies on five open source Python programs. We ran the test suites distributed with each of these programs using our tool to gather type information. We found that for these case studies the performance overhead of our monitoring technique was manageable, ranging from the same execution time to 2x - 5x slowdown. We also found that the accuracy and completeness of type information is dependent on the test suite coverage. Larger projects tended to provide larger test suites which produced better coverage.

6.7.1 Case study targets

We selected five open-source Python program from Github. The programs have various sizes and domains to represent different uses of Python. As we mentioned in the previous chapter, there is no standard benchmark for conducting empirical studies on Python programs which led us to search Github to select appropriate programs for our analysis. In addition, we selected larger programs than the previous chapter because we are performing an automated analysis instead of manual analysis and as a result we are no longer constrained by the size of the programs. We selected larger programs to reflect the size real-world applications — with the largest nearing 35000 LoC.

Table 6.6 summarises these programs and the number of lines of code without comments. They are:

ystockquote A small Python program which retrieves stock quote data from Yahoo Finance. This program illustrates the kind of scripts that are common in the Python community. The program is composed of one main file that contains three principal functions to query the Yahoo API. We modified it to add our type monitoring and run the test suite packed with the program.

expecter An assertion library for writing declarative assertions in unit tests and used by experienced developers. It is composed of one principal file containing several classes and methods. The library also provides several test suites. We modified the main file to add our type monitoring and then ran each test suite packaged with the program.

python-statlib A statistics library to showcase numerical programs. This library is composed of two main modules: stats.py which contains several statistics functions and pstat.py which contains common functions to manipulate lists. We modified each module to add our type monitoring and then ran the test suite packaged with the program.

webpy A web framework that was previously used by Reddit to showcase server applications. It is composed of one principal folder containing several modules responsible for different components such as the main application server, session management, form handling and various utilities. The program includes a test suite. We modified each module to add our type monitoring and then ran the test suite.

python-toolbox A collection of Python utilities for various tasks such as working with files, caching, data structures and manipulating iterables. It is composed of several modules which come each with a test suite.

Application	Description	LoC
ystockquote	Finance script	373
expecter	Testing library	430
python-statlib	Numerical library	5134
webpy	Web framework	7667
python-toolbox	Utilities	34511

Figure 6.6: Summary of case study targets

6.7.2 Methodology

As described at the beginning of the chapter, we developed *pytypemonitor* to provide insights about a program to help developers annotate functions with type information. We argued for a tool having two characteristics. First, the tool should be lightweight to allow simple integration in the developer workflow. Second, the tool should produce sufficient insights about the program to alleviate manual type annotation for programmers. To gather evidence, we conducted the following process:

1. Analyse the number of function and method definitions in each program
2. Run the test suite of the program and record the execution time. We record the average of five runs.
3. Add monitoring capability to the program
4. Run the test suite again and record the execution time. We record the average of five runs
5. Log type information about call instances in a separate file
6. Perform amalgamation of type information
7. Report results

6.7.3 Results

We now describe the results of running our tool on the selected Python programs. Figure 6.7 reports the data.

First, the results indicate that the tool's performance overhead is dependent on the the number of call instances. This is expected as each call instance is wrapped with the monitoring capability. The results collected shows that despite this, the performance overhead is manageable ranging from the same execution time to 2x - 5x the normal execution time. Second, we found the insights produced by our tool are dependent on the coverage of the testing suite. For example, only 3 functions in *ystockquote* when executed out of 84 function definitions, which is only a 4% coverage. This is expected given that our analysis is dynamic and produce results based on the program execution. Nonetheless, manual inspection of *ystockquote* shows that the other functions are only simple helper getter functions and the 3 functions monitored are indeed the main public-facing functions. In comparison, the analysis *webpy* produced a coverage of 83% and python-toolbox a coverage of 96%.

Metric	ystockquote	expecter	python-statlib	webpy	python-toolbox
# function defs	86	24	163	442	1815
# function defs monitored	3	15	19	370	1743
coverage ratio	4%	65%	12%	83%	96%
normal execution time	0.5s	0.1s	0.1s	4.8s	0.7s
monitored execution time	0.5s	0.1s	0.1s	9.3s	3.6s
# recorded call instances	3	121	47	96507	1095240

Figure 6.7: Summary of results

Analysis of python-toolbox We analysed python-toolbox in more detail to evaluate the practicality of insights produced by the reduction module. We found that the reduction phase significantly reduces the complexity of the output from the monitoring module.

We first applied the *remove duplicate*, *disjoint union of results* and *cartesian factorisation* phases because these phases are simple reductions that do not require a heuristic. We found that the vast majority of call instances were duplicate. In fact, the reduction phase only left 5193 call instances, which is 0.45% of the original set of call instances. Next, the disjoint union of results and exceptions types factorised the set further to 4802 call instances, which is a reduction of 7% of the previous phase. From this resulting set we found that 19% of call instances described unique function definitions and 81% were two or more call instances for a single function definition. We found that the cartesian factorisation was not frequently applicable. In fact, only one function definition met the criteria. It is called `cute_divmod` in the `math_tools.misc` package and returns the division and modulo for two numbers as a tuple:

```
def cute_divmod(a: float, b: int)  ->  tuple
def cute_divmod(a: int, b: float)  ->  tuple
def cute_divmod(a: int, b: int)  ->  tuple
def cute_divmod(a: float, b: float)  ->  tuple
```

The above function type signatures were simplified to:

```
def cute_divmod(a: int or float, b: int or float)  ->  tuple
```

Next, we analysed the result of the factorisation by type parameter and factorisation by supertype phases.

By applying the factorisation by type parameter, we found three related function definitions in the testing framework packaged in python-toolbox that were candidates. The method _baseAssertEqual, _getAssertEqualityFunc are method helpers for the method assertEqual. All these three methods check whether two argument are equal and could be simplified to use a type parameter:

```
def _baseAssertEqual[T](a: T, b: T)
def _getAssertEqualityFunc[T](a: T, b: T)
def assertEqual[T](a: T, b: T)
```

Using this reduction 82 call instances could be reduced to three signatures, representing 2% of the set of call instances after the disjoint union phase.

Finally, we found that common supertype elimination helps simplify signatures further. We opted for an approximate approach and used the supertype (object excluded) if four or more subtypes were available. We found one method called __exit_using_manage_context produced 277 distinct call instances that could be simplified to one function. We manually inspected its definition and its parameter names confirmed the validity of this grouping:

```
__exit_using_manage_context(self, exc_type, exc_value, exc_traceback):
```

The argument exc_type is always of type type, the argument exc_value of type Exception and exc_tracback of type traceback.

Nonetheless, we found this reduction to be mostly useful with methods dealing with collections. Several call instances were operating on collections which all share iterable as supertype. For example, the method logic_max produced the following call instances:

```
python_toolbox.logic_tools.logic_max(set)  ->  list
python_toolbox.logic_tools.logic_max(list)  ->  list
python_toolbox.logic_tools.logic_max(listiterator)  ->  list
python_toolbox.logic_tools.logic_max(tuple)  ->  list
```

These call instances can be reduced to the following simpler signature where set, list, listiterator and tuple are all iterables.

```
python_toolbox.logic_tools.logic_max(Iterable)  ->  list
```

This reduction collapsed an additional 352 call instances, representing 7% of the set of call instances after the disjoint union phase

After applying the five phases we found that the average number of generated signatures per function definition was 2.5 signatures. This result indicates that out lightweight technique can provide useful insights for programmers for future maintainability but without overwhelming them with too many multiple generated signatures.

6.7.4 Limitations

We now describe two limitations of our tool: coverage of the programs analysed and the Python features that are not supported.

6.7.4.1 Coverage

First, because it relies on a dynamic analysis, functions which are never called will not be monitored. For example, in the ystockquote program, only 3 functions out of 86 were monitored because the test suite did not cover them. A static analysis tool would be able to pick them up. Consequently, programmers may want to first get insights through a dynamic analysis and complement it with a static analysis if necessary.

6.7.4.2 Features

Second, our system does not monitor object attributes and nested functions. One could monitor access to attributes in Python by intercepting the `__setattr__` and `__getattr__` methods of objects. These functions are called respectively when an attribute is assigned to and accessed. For example, the following code monitors type information on assignment to the attribute content of the `Box` object:

```
class Box(object):
    def __setattr__(self, name, value):
        print str(name) + ":" + str(type(value))
        self.__dict__[name] = value

if __name__ == "__main__":
    b = Box()
    b.content = 10
    print b.content
```

We did not consider attributes in our system because they can typically be exposed through "getter" functions. In addition, intercepting each attribute access results in additional runtime overhead. For similar reasons, our system does not support nested functions. In fact, nested functions are not as useful (for monitoring purposes) as top-level functions because they are called within the parent function. As we explained in the previous chapter, one could implement monitoring of nested functions using Python's introspection module. Nested functions can be accessed via a tuple of code objects within the parent functions. Nonetheless, the tuple is immutable. As a result, one would have to create a copy of the parent function with an updated tuple of code objects that contains a wrapped version of the nested function with monitoring capabilities. The implementation of this mechanism would result in additional runtime overhead and complexity which is why *pytypemonitor* does not support it. Finally, our systems does not monitor functions that are already decorated. This is because our tool uses the decorator mechanism to add monitoring to a function. Consequently, it would conflict with existing decorators on that function.

6.7.4.3 Call-site vs. Call-strings

Pytypemonitor records call-sites independently giving a set of call-sites. A possible extension to the monitoring process would be to record the *sequence* of call-sites including the types of input and output leading to a particular call-site (known as a *call-string*). This information would provide more accurate context which is useful for inferring parametric polymorphic signatures. Nonetheless we have noted in our evaluation in Section 5.8.2

that parametric polymorphism had limited scope in the Python programs we analysed beyond trivial collection processing functions. In addition, despite the additional accuracy that call-strings provide, they would lead to a foreign textual representation for type declarations. Consequently, it would hinder ease of understanding by developers familiar with type annotations as supported by mainstream programming languages.

6.8 Use Cases

We describe three use cases for our tool. First, pytypemonitor can be used to detect API changes within a program introduced by two different version of Python or with different versions of a module. Second, it can be used as a form of regression testing when used in combination with our tool pytypedecl. Third, pytypemonitor can generally be used as a program comprehension tool for generating documentation about a new code base which developers can then read to inspect particular code locations of interest.

6.8.1 Detecting API changes

Our tool can be used to detect API changes by:

1. Run pytypemonitor on the program using its test suites to generate type declarations used as API contracts

2. Run pytypemonitor on the program using a different language version

3. Perform a diff between the two sets of generated API contracts to locate possible incompatibilities

We conducted the following experiment on the *ystockquote* programme. Running it using Python 2.7 produced the following signature for the method `_request()` which is responsible for fetching data for a specific stock:

```
_request(str, str) -> unicode
```

Running the same tests using Python 3 produced a different signature, indicating a change occurred to the API. In fact, the method `_request()` returned `unicode` in Python 2.7 but now returns `str` using Python 3. These two objects have different semantics.

```
_request(str, str) -> str
```

After inspecting the body of `_request()` it was found that it is also dependent on the `urlopen` module which produces different result `bytes` or `str` depending on the different Python versions.

6.8.2 Regression testing

The goal of regression testing is to assert that changes introduced in code do not affect parts of the program that should continue to work correctly. It is typically performed by re-running test cases testing functionality of older features and ensuring that the program behaviour is the same.

Our tool *pytypemonitor* complements *pytypechecker* (described in the previous chapter) to provide a form of regression testing of APIs. In fact, to conduct regression testing, a set of contracts about the behaviour of the program is required to check against. The action is:

1. Run pytypemonitor on the program using its test suites to generate type declarations used as API contracts

2. Make changes to the program's implementation as new requirements are introduced

3. Enable runtime type checking by using pytypechecker and the generated type declarations

4. Run the program and collect assertions errors resulting from API breaking changes

Performing this workflow on the `ystockquote` program results in an immediate runtime error indicating that the old signature is violated.

6.8.3 Program comprehension

From a broader perspective, we see `pytypemonitor` as a useful tool for automatically generating documentation about the APIs of a particular program. Documenting code is seen as good software development practice. However, many developers do not like it as it is seen as time-consuming. Our tool can be used to produce concise type declarations to be used as a replacement for API documentation. This is useful for improving productivity when locating appropriate parts of the program that require further exploration.

6.9 Related work

Rubydust An et al. [31] presented a dynamic type inference algorithm and implementation for Ruby called Rubydust. In a nutshell, Rubydust's type inference algorithm introduces a type variable for each field, method parameter and return type and infers a ground type for them based on constraints produced using dynamic runs of the program. It is implemented by making use of proxy objects which wrap each field, method parameter and return type with the associated type variable. Constraints are solved offline once dynamic runs have been collected. Rubydust works as a library by making use of Ruby's introspection features. Their work was evaluated on small scale programs (from 96LoC to 750LoC). The authors report that the overhead of using proxy objects is quite high compared to the original program. For example, they report an overhead of up to 2800x the original running time. In comparison, our system reported a significantly lower overhead of 2x to 5x for programs 40x larger.

Starkiller Starkiller is a tool that compiles Python programs into equivalent C++ programs [114]. As part of the compilation process, it performs static type inference to generate optimised code. Starkiller's type inference is flow-insensitive and based on the Cartesian Product Algorithm [30].

In contrast with the traditional type-inference approach using unification [87, 67], the Cartesian Product Algorithm (CPA) infers types by statically modelling data flow. Concretely, for each function definition, the algorithm calculates the cartesian product of the possible type sets of each parameter. The type sets of each parameter is resolved at each call-site by looking up the data flow graph of each actual argument. In comparison to our work, pytypemonitor has several benefits over Starkiller. First, it is not restricted to a subset of Python features. For example, Starkiller does not support Python's dynamic features such as `eval` and `eval` which are permitted using pytypemonitor. Second, it uses a flow-insensitive analysis which may lead to loss of precision during type inference. By constrast, pytypemonitor always uses the possible execution paths because it perform the monitoring as the program is executed. Third, pytypemonitor caused only small overhead of 2x to 5x for medium sized programs which makes it practical to enable on production systems.

Mino Mino is a system that performs static type inference of Python programs [123]. However, Mino takes a different approach compared to traditional constraint-solving techniques. Mino sees type inference as a classification problem which can be solved using machine-learning techniques. Indeed, Mino relies on a training set of existing Python program which are already labelled with type information. Mino then uses this training set to infer the type of similar looking program structures. For example, variables named i are usually counters or index variables and can be inferred to be integers. In relation to our work, Mino has only reported this technique to infer built-in Python types such as `int`, `list`, `dict` and `tuple`.

JSTrace JSTrace is a tool for JavaScript which discovers type annotations for functions [113]. Rather than using a static analysis approach, JSTrace performs a dynamic analysis. The tool performs the analysis in two phases. First, the input program is preprocessed with additional instrumentation code to capture type information. Next, the modified input program is run to gather type information which are then used to generate concrete type annotations. The author reports that this approach did not impact performance when used with interactive user applications.

It is interesting to point out that the idea of tracing types emitted by a dynamically-typed program at runtime is not new. This idea has been used by just-in-time compilers to optimise the execution of dynamically-typed languages. Type traces are used to generate type-specialised native code [61]. In addition, it is also used as a way to supplement static type inference [59, 72].

Dynamic behaviour of Python applications Static analysis of Python code has been explored by several previous work including Starkiller described above. However, they typically restrict Python's expressiveness by assuming that most dynamic features such as reflection (e.g. adding a field at runtime) are not used in practice. To confirm these assumptions, Holkner et al. undertook a study that measures the dynamic behaviour of 24 open source Python programs. They found that reflective features were highly used in all Python programs. This strengthens our case for using dynamic type-inference as an alternative to static analysis.

Daikon system Daikon is a system that can infer likely program invariants such as whether a variable is in a certain range or is non-null [55]. Daikon works in three steps. First a given program is instrumented to write data trace files. Second, the instrumented program is run with a test suite. Finally, the invariant detector reads the data trace files and generates potential behavioural invariants. In comparison with our work, function type signatures can also be seen as a kind of program invariants. Nonetheless, our system directly monitors the behaviour of the program via proxy objects instead of a separate instrumentation phase. In addition, type information of a program are reduced to provide manageable function type signatures for programmers.

6.10 Conclusion

This chapter described *pytypemonitor*, a lightweight system that dynamically infers function type signatures for Python programs. We developed this system to help developers get insights about their programs. This system supports the typechecking mechanism we presented in the previous chapter by easing the number of manual type annotation required by developers. The architecture of the system we presented is extensible and applicable to other dynamically-typed languages. It consists of three components: collection of source code facts, amalgamation of source code facts and translation to a type declaration language. We demonstrated the use of proxy objects which monitor the input and output behaviour of functions as a collection mechanism. In addition, we presented five techniques which simplify the collected information to produce manageable function type signatures. In addition, we presented the implementation of our system and showed that the performance overhead of our monitoring mechanism was manageable. Finally, we evaluated our system on five Python programs and showed that it produce useful insights given sufficient coverage for larger programs.

Chapter 7

Conclusions

This chapter concludes this dissertation with an overall summary, overview of future work and final remarks.

7.1 Summary

This dissertation had two related aims. First, we argued that programming language evolution is a complex and important subject that affects both language designers and developers. Second, this thesis made the argument that programming language evolution can be supported with program comprehension tools.

To this end, this dissertation started by characterising the multiple factors driving programming language evolution. It then established a classification of incompatible evolution changes. Finally, we developed three instances of tools to provide insights about programming language evolution. We provided supporting evidence and evaluation for these tools. We found that the techniques we developed are lightweight and simple to integrate in large programs and can diagnose programming language evolution issues that would be difficult for developers to manually locate and understand.

In summary, this dissertation made the following contributions.

Chapter 2 characterised factors driving language evolution based on developer needs, influences from other programming language communities and business-driven agendas. Some material of this chapter was published in the book Java 8 in Action [125].

Chapter 3 argued that imcompatible programming language evolution is problematic for language designers and developers. To this end, this chapter established a classification of incompatible changes introduced into programming languages that affect developers. It provided a survey of use cases based on an analysis of compatibility guides and bug reports from several programming languages.

Chapter 4 contributed a search-based system that lets language designers and developer locate relevant code patterns in large code bases. This work was published in Science of Computer Programming [127].

Chapter 5 addressed the problem that certain classes of incompatibilities are difficult to locate accurately in dynamically typed programming languages. It contributed a lightweight run-time type-checking system for Python which let programmers annotate their code to produce diagnosis and explanations about the location of these incompatibilities. This work influenced the design of a standard type annotation language as part of a Python 3 enhancement proposal by Rossum et al [132].

Chapter 6 contributed a dynamic analysis to automatically provide migration insights about a Python program. This analysis generated function type signatures to alleviate the burden of manual annotation for programmers.

7.2 Further work

We hope the work presented in this dissertation opens new research avenues for programming language evolution. This section suggests further research in the following areas.

7.2.1 A Corpus querying platform

The source-code querying system we presented in Chapter 4 can be extended into a corpus querying platform to help language design decisions. In fact, there are several proposals such as Qualitas Corpus (a corpus of open source software) that aim to help language researchers examine code [120]. A successful platform requires a user-friendly source code query language that is close to the original source code programming language. Such queries could be translated to queries in our system. Our system could also be extended with support for version history [118]. This would enable researchers to ask a wider range of questions such as adoption of certain development practices, new libraries or language features over time. Finally, our system stores as overlays those relations corresponding to compiler data structures, such as control-flow graph and call graph. However, this is rather ad-hoc and inefficient. Ideally, we would like overlays to be seen as cached results of queries. This would be very attractive, in that a user query may then be defined as pre-calculating and storing overlay relations before the query is evaluated.

7.2.2 Contracts for programming language evolution

The work we presented with Python in Chapter 5 and 6 made trade-offs in terms of expressiveness and simplicity. For example, it focused on type-level information and did not support annotations for local variables and nested functions. In addition, it is implemented as a runtime analysis. An interesting extension would be to design a hybrid approach which allows both static type checking and dynamic type checking while also simply integrating with existing toolchains. This would enable our system to minimise the time spent on runtime analysis. Another interesting direction is to apply this technique to other level of incompatibilities which we presented in Chapter 2 such as behavioural and security incompatibilities.

7.2.3 Refactoring

This dissertation focused on providing search support in the context of programming language evolution. Further work is required to integrate search techniques with refactoring approaches. In fact, another problematic aspect for developers is the manual code changes required after identifying code location of interest. This idea has already been applied in the context of upgrading code bases which use deprecated idioms. However, this mechanism could also be applied to automatically fixing incompatibilities when upgrading to a newer language version. For example, a search tool may identify source code locations that introduce incompatibilities. Another tool could make code transformation suggestions or fix the incompatibilities if possible. Such a tool could then be provided as part of the language compiler toolchain.

7.3 Final remarks

In closing, we hope this dissertation will spur further interest in establishing programming language evolution as a research field of its own. We have argued in this dissertation that programming language evolution is a cross-disciplinary subject that can benefit not only from academic research in programming language and software engineering but also from linguistics and social and economic research. In addition, it is crucial that industrial needs are taken in consideration because companies and developers are the first to suffer from language evolution. To this end, it is essential that academic research and industry work together for the benefit of all users of programming languages.

Bibliography

[1] A Scala Corrections Library. http://www.slideshare.net/extempore/a-scala-corrections-library.

[2] BBQ. http://browsebyquery.sourceforge.net/.

[3] C# programming language future features : Iterators. http://msdn.microsoft.com/en-us/library/aa289180%28v=vs.71%29.aspx#vbconcprogramminglanguagefuturefeaturesanchor3.

[4] Ceylon. http://ceylon-lang.org/.

[5] Changes in Java SE 6 update 17. http://www.oracle.com/technetwork/java/javase/6u17-141447.html.

[6] Coverity White Paper: The Next Generation of Static Analysis. http://www.coverity.com/library/pdf/Coverity_White_Paper-SAT-Next_Generation_Static_Analysis.pdf.

[7] Eclipse Java development tools. http://www.eclipse.org/jdt.

[8] Findbugs. http://findbugs.sourceforge.net.

[9] The for-each loop. http://docs.oracle.com/javase/1.5.0/docs/guide/language/foreach.html.

[10] Google code search. http://code.google.com/codesearch.

[11] Jackpot. http://wiki.netbeans.org/Jackpot.

[12] Java 8: support for more aggressive type-inference. http://mail.openjdk.java.net/pipermail/lambda-dev/2012-August/005357.html.

[13] Java coding standard. http://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_Java.pdf.

[14] JEP 107: Bulk data operations for collections. http://openjdk.java.net/jeps/107.

[15] JEP 126: Lambda expressions & virtual extension methods. http://openjdk.java.net/jeps/126.

[16] JSR 201: Extending the Java programming language with enumerations, autoboxing, enhanced for loops and static import. https://jcp.org/ja/jsr/detail?id=201.

[17] Junit: sort test methods for predictability. https://github.com/junit-team/junit/pull/293.

[18] Neo4j Graph Database. http://www.neo4j.org/.

[19] Object Query Language. http://www.ibm.com/developerworks/java/library/j-ldn1.

[20] Oracle vs. google. http://www.cafc.uscourts.gov/images/stories/opinions-orders/13-1021.Opinion.5-7-2014.1.PDF.

[21] PMD. http://pmd.sourceforge.net/.

[22] Proposal: improved type inference for generic instance creation. http://mail.openjdk.java.net/pipermail/coin-dev/2009-February/000009.html.

[23] Pros and cons about Python 3. http://lucumr.pocoo.org/2010/1/7/pros-and-cons-about-python-3/.

[24] PyCharm types. https://github.com/JetBrains/python-skeletons#types.

[25] Random number bug in Debian Linux. https://www.schneier.com/blog/archives/2008/05/random_number_b.html.

[26] Scala migration manager. https://github.com/typesafehub/migration-manager.

[27] SQLmap issue: port code to Python 3. https://github.com/sqlmapproject/sqlmap/issues/93.

[28] Why Jetbrains needs Kotlin. http://blog.jetbrains.com/kotlin/2011/08/why-jetbrains-needs-kotlin/.

[29] Wiggle Project. http://www.urma.com/wiggle.

[30] Ole Agesen. The cartesian product algorithm. In *ECOOP95, Aarhus, Denmark, August 7–11, 1995*, pages 2–26. Springer, 1995.

[31] David An, Avik Chaudhuri, Jeffrey Foster, and Michael Hicks. Dynamic inference of static types for Ruby. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)*, pages 459–472. ACM, 2011.

[32] Jong-hoon David An, Avik Chaudhuri, Jeffrey S Foster, and Michael Hicks. *Dynamic inference of static types for Ruby*, volume 46. ACM, 2011.

[33] R. Angles and C. Gutierrez. Survey of graph database models. *Computing Surveys*, 40(1):1, 2008.

[34] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[35] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An infrastructure for the large-scale collection and analysis of open-source code. *Science of Computer Programming*, 2012.

[36] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science and Engineering*, 13.2:31–39, 2011.

[37] Sotirios Beis, Symeon Papadopoulos, and Yiannis Kompatsiaris. Benchmarking graph databases on the problem of community detection. In *New Trends in Database and Information Systems II*, pages 3–14. Springer, 2015.

[38] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, February 2010.

[39] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012.

[40] Shawn A Bohner. Software change impact analysis. 1996.

[41] Bracha, Gilad, et al. JSR 14: Add generic types to the Java programming language. `http://www.jcp.org/en/jsr/detail`, 1999.

[42] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. *ACM SIGPLAN Notices*, 33(10):183–200, 1998.

[43] Silvia Breu and Thomas Zimmermann. Mining aspects from version history. In Sebastian Uchitel and Steve Easterbrook, editors, *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*. ACM Press, September 2006.

[44] Alex Buckley. Different kinds of compatibility. 2007.

[45] Tal Cohen, Joseph (Yossi) Gil, and Itay Maman. JTL: The Java tools language. In *OOPSLA*, 2006.

[46] Keith D Cooper, Philip J Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. *ACM SIGPLAN Notices*, 34(7):1–9, 1999.

[47] William Croft. Evolutionary linguistics. *Annual Review of Anthropology*, 37(1):219, 2008.

[48] Joseph D. Darcy. Kinds of compatibility: Source, binary, and behavioral. 2008.

[49] O. de Moor, M. Verbaere, and E. Hajiyev. Keynote address: .QL for source code analysis. In *SCAM*, 2007.

[50] Coen De Roover, Carlos Noguera, Andy Kellens, and Vivane Jonckers. The SOUL tool suite for querying programs in symbiosis with Eclipse. In *PPPJ*, 2011.

[51] Molisa Derk. What makes a programming language popular?: an essay from a historical perspective. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pages 163–166. ACM, 2011.

[52] Jens Dietrich, Kamil Jezek, and Premek Brada. What Java developers know about compatibility, and why this matters. *Empirical Software Engineering*, pages 1–26, 2015.

[53] Sophia Drossopoulou, David Wragg, and Susan Eisenbach. What is Java binary compatibility? In *ACM SIGPLAN Notices*, volume 33, pages 341–361. ACM, 1998.

[54] Robert Dyer, Hoan Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *ICSE*, 2013.

[55] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[56] Mike Fagan. *Soft typing: an approach to type checking for dynamically typed languages*. PhD thesis, Citeseer, 1992.

[57] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN Notices*, volume 37, pages 48–59. ACM, 2002.

[58] Python Software foundation. Python success stories. `https://www.python.org/about/success/`.

[59] Michael Furr, Jong-hoon David An, and Jeffrey S Foster. Profile-guided static typing for dynamic scripting languages. *ACM SIGPLAN Notices*, 44(10):283–300, 2009.

[60] 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*, pages 1859–1866. ACM, 2009.

[61] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. In *ACM Sigplan Notices*, volume 44, pages 465–478. ACM, 2009.

[62] J. Gil and K. Lenz. The use of overloading in Java programs. In *ECOOP*, 2010.

[63] Briant Goetz. Language designer's notebook: Quantitative language design. 2010.

[64] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.

[65] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *Java 8 Language Specification*. Oracle, 2014.

[66] E. Hajiyev, M. Verbaere, and O. De Moor. Codequest: Scalable source code queries with datalog. *ECOOP 2006–Object-Oriented Programming*, pages 2–27, 2006.

[67] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, pages 29–60, 1969.

[68] Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1435–1441. ACM, 2006.

[69] American National Standards Institute. *Rationale for the ANSI C Programming Language*. Silicon Press, Summit, NJ, USA, 1990.

[70] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *AOSD*, 2003.

[71] Salim Jouili and Valentin Vansteenberghe. An empirical comparison of graph databases. In *Social Computing (SocialCom), 2013 International Conference on*, pages 708–715. IEEE, 2013.

[72] Madhukar N Kedlaya, Jared Roesch, Behnam Robatmili, Mehrdad Reshadi, and Ben Hardekopf. Improved type specialization for dynamic scripting languages. In *Proceedings of the 9th Symposium on Dynamic Languages*, pages 37–48. ACM, 2013.

[73] A. Kellens, C. De Roover, C. Noguera, R. Stevens, and V. Jonckers. Reasoning over the evolution of source code using quantified regular path expressions. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 389–393. IEEE, 2011.

[74] Donald Ervin Knuth and Luis Trabb Pardo. *The early development of programming languages*. Stanford University, Computer Science Department, 1976.

[75] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology — CRYPTO96*, pages 104–113. Springer, 1996.

[76] Sameer Kulkarni and John Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. *ACM SIGPLAN Notices*, 47(10):147–162, 2012.

[77] Peter J Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[78] Jukka Lehtosalo. *Adapting dynamic object-oriented languages to mixed dynamic and static typing*. PhD thesis, University of Cambridge, 2014.

[79] Martin Madsen, Peter Sørensen, and Kristian Kristensen. *Ecstatic–type inference for Ruby using the cartesian product algorithm*. PhD thesis, Master Thesis, Jun, 2007.

[80] Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In *ECOOP 2008–Object-Oriented Programming*, pages 260–284. Springer Berlin Heidelberg, 2008.

[81] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A Bader. A performance evaluation of open source graph databases. In *Proceedings of the first workshop on Parallel programming for analytics applications*, pages 11–18. ACM, 2014.

[82] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706. ACM, 2006.

[83] Robin Message. Programming for humans: a new paradigm for domain-specific languages. Technical Report UCAM-CL-TR-843, University of Cambridge, Computer Laboratory, November 2013.

[84] Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

[85] Leo A Meyerovich and Ariel S Rabkin. Socio-plt: Principles for programming language adoption. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, pages 39–54. ACM, 2012.

[86] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In *ECOOP98Object-Oriented Programming*, pages 355–382. Springer, 1998.

[87] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[88] Oracle. Reflection should not allow final fields to be modified. `https://bugs.openjdk.java.net/browse/JDK-4250960`, 1999.

[89] Oracle. Java object serialization specification. `http://docs.oracle.com/javase/6/docs/platform/serialization/spec/serialTOC.html`, 2005.

[90] Oracle. JDK-6480539 : BigDecimal.stripTrailingZeros(). `http://bugs.java.com/bugdatabase/view_bug.do?bug_id=6480539`, 2006.

[91] Oracle. Java SE 7 and JDK 7 compatibility. `http://www.oracle.com/technetwork/java/javase/compatibility-417013.html`, 2011.

[92] Oracle. Remove offset and count fields from java.lang.String. `http://mail.openjdk.java.net/pipermail/core-libs-dev/2012-June/010509.html`, 2012.

[93] Oracle. Handle frequent HashMap collisions with balanced trees, 2013.

[94] Oracle. Improve copying behaviour of String.subSequence(). `http://mail.openjdk.java.net/pipermail/core-libs-dev/2013-February/014609.html`, 2013.

[95] Oracle. String.substring and String.subsequence performance slower since 7u6. `https://bugs.openjdk.java.net/browse/JDK-7197183`, 2013.

[96] Oracle. Compatibility guide for JDK 8. `http://www.oracle.com/technetwork/java/javase/8-compatibility-guide-2156366.html#A999198`, 2014.

[97] Dominic Orchard and Andrew Rice. Upgrading fortran source code using automatic refactoring. In *Proceedings of the 2013 ACM workshop on Workshop on refactoring tools*, pages 29–32. ACM, 2013.

[98] Jeffrey L Overbey and Ralph E Johnson. Regrowing a language: refactoring tools allow programming languages to evolve. In *ACM SIGPLAN Notices*, volume 44, pages 493–502. ACM, 2009.

[99] Jeffrey L Overbey, Stas Negara, and Ralph E Johnson. Refactoring and the evolution of fortran. In *Software Engineering for Computational Science and Engineering, 2009. SECSE'09. ICSE Workshop on*, pages 28–34. IEEE, 2009.

[100] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E Hassan. What do programmers know about the energy consumption of software? *PeerJ PrePrints*, 3:e1094, 2015.

[101] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 3–12. ACM, 2011.

[102] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *Mining Software Repositories*, 2011.

[103] S. Paul and A. Prakash. A query algebra for program databases. *Software Engineering, IEEE Transactions on*, 22(3):202–217, 1996.

[104] Benjamin C Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, 1994.

[105] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[106] Peter Pirkelbauer, Damian Dechev, and Bjarne Stroustrup. Source code rejuvenation is not refactoring. In *SOFSEM 2010: Theory and Practice of Computer Science*, pages 639–650. Springer, 2010.

[107] Peter Mathias Pirkelbauer. *Programming language evolution and source code rejuvenation*. PhD thesis, Texas A&M University, 2010.

[108] Brianna M Ren, John Toman, T Stephen Strickland, and Jeffrey S Foster. The Ruby type checker. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1565–1572. ACM, 2013.

[109] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. In *ACM Sigplan Notices*, volume 39, pages 432–448. ACM, 2004.

[110] John C Reynolds. The coherence of languages with intersection types. In *Theoretical aspects of computer software*, pages 675–700. Springer, 1991.

[111] R Robbes, E Tanter, and D Rothlisberger. How developers use the dynamic features of programming languages: the case of smalltalk. In *Proceedings of the International Working Conference on Mining Software Repositories*, 2011.

[112] Barbara G Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53. ACM, 2001.

[113] Claudiu Saftoiu. *JSTrace: Run-time type discovery for JavaScript.* PhD thesis, Masters thesis, Brown University, 2010.

[114] Michael Salib. *Starkiller: A static type inferencer and compiler for Python.* PhD thesis, MIT, 2004.

[115] Herbert Schildt. *C# 4.0: The complete reference.* McGraw-Hill, 2010.

[116] Jeremy Siek and Walid Taha. Gradual typing for objects. In *ECOOP 2007–Object-Oriented Programming*, pages 2–27. Springer, 2007.

[117] TIOBE Software. TIOBE index september 2014. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.

[118] Reinout Stevens. Source Code Archeology using Logic Program Queries across Version Repositories. Master's thesis, Vrije Universiteit Brussel, 2011.

[119] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[120] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The qualitas corpus: A curated collection of java code for empirical studies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 336–345. IEEE, 2010.

[121] Satish Thatte. Quasi-static typing. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 367–381. ACM, 1989.

[122] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, 2009.

[123] Stephen Tu. Mino: Data-driven approximate type inference for python. 2010.

[124] Raoul-Gabriel Urma and Mario Fusco. From imperative programming to fork/join to parallel streams in Java 8. `http://www.infoq.com/articles/forkjoin-to-parallel-streams`, 2014.

[125] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft. *Java 8 in Action.* Manning, 2014.

[126] Raoul-Gabriel Urma and Jonathan Gibbons. Java Compiler Plug-ins in Java 8. *Oracle Java Magazine*, 2013.

[127] Raoul-Gabriel Urma and Alan Mycroft. Source-code queries with graph databaseswith application to programming language usage and evolution. *Science of Computer Programming*, 2013.

[128] Raoul-Gabriel Urma and Janina Voigt. Using the OpenJDK to investigate covariance in Java. *Oracle Java Magazine*, 2012.

[129] R.G. Urma and A. Mycroft. Programming language evolution via source code query languages. In *Proceedings of the ACM 4th annual workshop on Evaluation and usability of programming languages and tools*, pages 35–38. ACM, 2012.

[130] Guido van Rossum. Whats new in Python 3.0. `https://docs.python.org/3/whatsnew/3.0.html`, 2008.

[131] Guido van Rossum and Raymond Hettinger. PEP 308 - conditional expressions. `http://legacy.python.org/dev/peps/pep-0308/`, 2003.

[132] Guido van Rossum, Jukka Lehtosalo, and Lukasz Langa. PEP 484 - type hints. `https://www.python.org/dev/peps/pep-0484/`, 2014.

[133] Steven R Vegdahl. Phase coupling and constant generation in an optimizing microcode compiler. In *ACM SIGMICRO Newsletter*, volume 13, pages 125–133. IEEE Press, 1982.

[134] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th annual Southeast regional conference*, page 42. ACM, 2010.

[135] Michael M. Vitousek, Andrew Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for python. In *DLS*, 2014.

[136] Deborah L Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):1053–1084, 1997.

[137] Dachuan Yu, Zhong Shao, and Valery Trifonov. Supporting binary compatibility with static compilation. In *Java Virtual Machine Research and Technology Symposium*, pages 165–180, 2002.