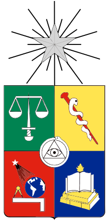

UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

EMPIRICALLY-DRIVEN DESIGN AND IMPLEMENTATION OF GRADUALTALK

TESIS PARA OPTAR AL GRADO DE DOCTOR EN CIENCIAS
MENCIÓN COMPUTACIÓN

OSCAR EDWIN ALVAREZ CALLAÚ

PROFESORES GUÍAS:
ÉRIC TANTER
ROMAIN ROBBES

MIEMBROS DE LA COMISIÓN:
MARÍA CECILIA BASTARRICA PIÑEYRO
ALEXANDRE BERGEL
OSCAR NIERSTRASZ

Este trabajo ha sido parcialmente financiado por CONICYT, NIC Chile y Microsoft Research

SANTIAGO DE CHILE
2015

Resumen

Los lenguajes de tipado dinámico permiten un desarrollo ágil. Sin embargo, cuando estos pequeños programas se convierten en aplicaciones grandes, depurar se vuelve una tarea tediosa. Esto se debe principalmente a que los errores son solo detectables en tiempo de ejecución. Smalltalk, al ser un lenguaje de tipado dinámico, sufre de estos problemas. Los sistemas de tipos pueden disminuir ciertos errores de los lenguajes de tipado dinámico. Además, la inserción de tipos mejora la documentación de APIs, provee un mejor soporte a los editores y ayuda a optimizar la compilación. Los sistema de tipos, especialmente diseñados para lenguajes existentes, son llamados sistema de tipos retro-alimentados (*retrofitted type systems* en inglés).

Diseñar un sistema de tipos retro-alimentado es una tarea complicada. Esto se debe a que tales sistemas de tipos deben soportar patrones de programación muy particulares (llamados idioms), minimizar la refactorización de código por la inserción de tipos, y proveer una integración entre las partes (ej. módulos) con y sin tipos. Estos problemas son exacerbados cuando el lenguaje destino es altamente dinámico, como Smalltalk. Si bien se ha intentado insertar tipos en Smalltalk, ellos no han sido diseñados como sistemas de tipos retro-alimentados.

En este trabajo de tesis presentamos Gradualtalk, un sistema de tipos retro-alimentado para Smalltalk, que soporta la mayoría de las características particulares e idioms de Smalltalk. En la parte del diseño, nosotros analizamos detalladamente cual es el mejor sistema de tipos gradual y aquellas extensiones que mejor encajan en Gradualtalk. Cada una de estas extensiones son claramente justificadas usando evidencia (empírica) disponible en la literatura o propuesta por nosotros. En detalle, nosotros presentamos como evidencia empírica dos estudios a larga escala sobre las características dinámicas de Smalltalk y sobre los predicados de tipos. Ademas presentamos tres estudios preliminares sobre el uso de `self`, el uso de variables que pueden representar varios valores de diferente tipo, y el uso de colecciones. Con toda esta información implementamos una primera versión de Gradualtalk. Finalmente, validamos Gradualtalk mediante la inserción de tipos en varios proyectos Smalltalk reales.

Abstract

Dynamically typed languages allow for agile development. However, when these little scripts become large programs, debugging turns into a tedious task. This is mainly because errors are only noticed at run time. Smalltalk, as a dynamically typed language, suffers from such problems. Type systems can lessen the error proneness of dynamically typed languages, because some kinds of errors can be detected at compile time. Furthermore, the introduction of types improves API documentation, provides better IDE-related tools support (*e.g.* code navigation and completion), and maximizes compiler optimizations. These type systems, especially targeted to existing dynamically typed languages, are called retrofitted type systems.

Developing retrofitted type systems is difficult, in particular its design part. This is because such type systems must support particular programming idioms, reduce refactoring due to type restrictions, and provide a seamless integration between typed and untyped code. These issues are even exacerbated in highly-dynamic languages, such as Smalltalk, due to several factors that make Smalltalk unique: its particular features, *e.g.* traits and open classes; a powerful reflective API; and its live environment. Although there have been some attempts to bring types to Smalltalk, they were not designed as retrofitted type systems.

In this thesis work, we present Gradualtalk, a retrofitted type system for Smalltalk that covers most (if not all) Smalltalk features and idioms. In the design, we carefully discuss both the best partial type system for Gradualtalk, and the most suitable typing extensions. Each typing extension has been properly justified with (empirical) evidence from the literature (when available) and Smalltalk-specific empirical studies performed by us: two large-scale studies on the use of dynamic features, and the use of type-based dispatch patterns; and three preliminary studies on the use of `self` as a return value, the use of variables that may represent different types, and the use of collections. We implement a first version of Gradualtalk based on the feedback and decisions from the design. Additionally, we validate Gradualtalk through typing several Smalltalk projects, and report on bugs, refactoring and typing challenges that we found in that process.

To my family.

A mi familia.

Agradecimientos

Ante todo quiero agradecer a Dios por todo el apoyo brindado desde un principio en mi vida hasta este gran momento. Y tengo fe de que continuara apoyando me. El trabajo de esta tesis ha tenido el apoyo de innumerables personas y organizaciones. Es muy difícil nombrar a todos en una sola página.

Primeramente quisiera agradecer a mi familia por todo el apoyo incondicional que he tenido en todos estos años. Mi esposa Caren Vargas, mi hijo Santiago, mis mamás María Jesús Callaú y María Rosita Callaú, mi tío Armando Rocha, mi hermano Marco y demás familiares que siempre han estado presentes. Sin ellos no hubiera podido terminar este trabajo, ni ser la persona que soy.

Sin lugar a duda mis tutores Éric Tanter y Romain Robbes han hecho un trabajo titánico en enseñarme lo que es ser un investigador profesional. Siempre estaré en deuda con ellos, ya que sin su apoyo, enseñanzas y buena voluntad, no hubiera podido terminar este trabajo ni conocer el mundo de la investigacin. De igual forma agradezco mucho a mi comisión: Cecilia Bastarrica, Alex Bergel y Oscar Nierstrasz. Sus comentarios ha mejorado mi trabajo de tesis. También quiero agradecer a los profesor del DCC que han contribuido a mi formación profesional, especialmente al profesor Claudio Gutiérrez.

Estoy muy agradecido con Chile, mi segunda patria, ya que por medio de CONICYT he podido financiar parte de mi doctorado. Similarmente agradezco a NIC Chile (DCC) por brindarme apoyo económico cuando más lo necesitaba. De igual forma estoy muy agradecido con Microsoft Research, especialmente Tom Zimmermann, Nachi Nagappan y Jaime Puente, por creer en mi y en mi investigación. Sin el apoyo financiero de estas instituciones no hubiera podido comenzar, avanzar o terminar este trabajo.

Finalmente quiero agradecer a todas las personas del DCC y FCFM que han colaborado en mi vida de estudiante de doctorado a cada nivel de esta. Angélica Aguirre y Sandra Gáez siempre ha estado dispuestas a facilitar mi vida estudiantil. Un especial agradecimiento a mis compañeros de laboratorio y estudios: Guillaume Pothier, Paul Leger, Ismael Figueroa, Esteban Allende, Juan Pablo Sandoval, Alcides Quispe y Milton Mamami entre muchos otros.

Table of Contents

List of Tables x

List of Figures xii

1 Introduction 1

2 Type Systems and Smalltalk 7

2.1 Static type systems . . . 7
2.2 Dynamic type systems . . . 9
2.3 Partial type systems . . . 10
2.4 Retrofitted type systems for dynamic languages . . . 14
2.5 Type systems for Smalltalk . . . 19
2.6 Problem statement . . . 21

Part I: Empirically-Driven Design 23

3 Designing A Retrofitted Type System 24

3.1 Introduction . . . 24
3.2 The Smalltalk language . . . 26
3.2.1 Smalltalk core . . . 26
3.2.2 Smalltalk features . . . 27
3.2.3 Programming idioms . . . 30
3.3 Type system goals and foundations . . . 33
3.3.1 Type system goals . . . 33
3.3.2 Foundations . . . 35
3.4 Type system features . . . 37

3.4.1 Features and idioms covered by the base type system 38
3.4.2 Features and idioms covered by a typing feature 38
3.4.3 Summary 46

4 Preliminary Empirical Studies 47
4.1 Introduction 47
4.2 Experimental setup 49
4.3 On the use of `self` as a return value 50
4.3.1 Methodology 51
4.3.2 Results and discussion 52
4.4 On the use of joining values 53
4.4.1 Methodology 54
4.4.2 Results and discussion 54
4.5 On the use of collections 56
4.5.1 Methodology 57
4.5.2 Results and discussion 59
4.6 Threats to validity 61
4.7 Conclusions 62

5 How and Why Developers Use the Dynamic Features of Smalltalk 64
5.1 Introduction 65
5.2 Experimental setup 68
5.2.1 Methodology 68
5.2.2 Project categories 70
5.2.3 Analyzed dynamic features 70
5.3 Quantitative Results 75
5.3.1 How programmers use dynamic features 75
5.4 How each dynamic feature is used 79
5.5 Discussion 92
5.6 Why do developers resort to using dynamic features? (and what to do about it) 94
5.6.1 Methodology 95
5.6.2 Categorizing user intention when using dynamic features . . 98

5.6.3 Types of applications 107
5.6.4 Summary 109
5.7 Threats to validity 110
5.8 Related work 112
5.9 Conclusions 115

6 On the Use of Type Predicates in Smalltalk 117
6.1 Introduction 118
6.2 Experimental setup 121
6.2.1 Corpus 121
6.2.2 Finding predicates and their usages 122
6.3 Prevalence of type predicates 124
6.3.1 Basic statistics in Squeaksource 124
6.3.2 Usage categories 125
6.3.3 Refinement 127
6.3.4 Prevalence of predicate usages 128
6.3.5 Summary 130
6.4 Prevalence of categories of type predicates 130
6.4.1 Predicate categories 130
6.4.2 Usage contexts and predicate categories 131
6.4.3 Nil predicate 132
6.4.4 Polymorphic predicates 133
6.4.5 Summary 133
6.5 Prevalence of logical combinations 133
6.5.1 Overall prevalence of logical combinations 134
6.5.2 Prevalence in nil predicates 134
6.5.3 Nominal and polymorphic predicates 134
6.5.4 Structural predicates 135
6.5.5 Summary 135
6.6 Prevalence of constant predicates 135
6.6.1 Classification of predicates 136
6.6.2 Prevalence of constant predicates 138
6.6.3 Relevance of predicate names 138

6.6.4 Relationship between constancy and usage 139
6.6.5 Dynamic analysis of predicates 139
6.6.6 Summary 141
6.7 Threats to validity 141
6.8 Related work 145
6.9 Conclusion 148

Part II: Gradualtalk **150**

7 Introduction to Gradualtalk **151**
7.1 From dynamically typed to gradually typed code 151
7.2 Closures 153
7.3 Self and metaclasses 153
7.4 Casts 154
7.5 Parametric polymorphism 155
7.6 Union types 156
7.7 Structural and nominal types 158
7.7.1 Structural types 158
7.7.2 Nominal types 160
7.7.3 Reconciling nominal and structural types 161
7.8 Live system 162
7.9 Gradualtalk static semantics 164
7.9.1 Types in Gradualtalk 165
7.9.2 Self types rules 165
7.9.3 Subtyping 166
7.9.4 Safety and type soundness 169

8 Gradualtalk Validation **170**
8.1 Corpus and methodology 170
8.2 Overview of findings 172
8.3 Bugs and refactoring 174
8.4 Interesting illustrations of Gradualtalk 176
8.5 Typing challenges 176
8.6 Threats to validity 180

8.7 Conclusions 181

Part III: Conclusions **184**

9 Contributions **185**

9.1 Gradualtalk 186

9.2 Empirical studies 187

10 Perspectives **190**

10.1 Gradualtalk 190

10.2 Empirical studies 191

Bibliography **195**

List of Tables

3.1 Smalltalk and Java syntax compared. 27
3.2 Proposed typing features for Gradualtalk that cover one or more language features. 39
3.3 A summary of covered language features and idioms by the base type system and typing features. 46

5.1 The 10 largest projects in our study. 69
5.2 Per-feature distribution of safe and unsafe calls, where unsafe calls are sorted by project category. In bold: category that is considerably over-represented (over-representation factor > 4) 80
5.3 Per-feature distribution of the sample set size 96

6.1 Usage categories of type predicates with their refinements. 126
6.2 Usages distributions for coarse and fine-grained predicate categories. 130
6.3 Usage contexts and predicate categories: The first group of three columns shows the number of usages by context and category. The second group shows the distribution of usage contexts by predicate categories (columns sum 100%). The last group shows the distribution of predicate categories by usage contexts (rows sum 100%). . . 131

7.1 Types in Gradualtalk 165

8.1 Projects typed with Gradualtalk, * indicates these are not all classes of the project. 172
8.2 Usage of types in methods (*mth*) and classes (*cls*). 173

8.3 Type relation for `Number >>> #+`. Rows correspond to the receiver type, columns correspond to the argument type and each cell value is the corresponding return type. 177

List of Figures

4.1 Method distribution based on their return. 53
4.2 Presence of joining values in projects, classes and methods. 55
4.3 Distribution of commented methods about collections. 59
4.4 Distribution of collection objects in average per inspection. 60

5.1 Distribution of dynamic feature usages. 77
5.2 Per-feature distribution of all projects arranged by category of use. 78
5.3 Safe/unsafe usages of instance creation. 81
5.4 Safe/unsafe usages of class creation. 81
5.5 Unsafe uses of object references updates. 82
5.6 Safe/unsafe usages of object field reads. 83
5.7 Safe/unsafe usages of object field updates. 83
5.8 Safe/unsafe usages of message sending. 84
5.9 Safe (green)/unsafe(red and yellow) usages of class deletion. 86
5.10 Safe/unsafe usages of superclass updates. 86
5.11 Safe/unsafe uses of method compilation. 87
5.12 Safe/unsafe uses of method removal. 88
5.13 Safe/unsafe uses of Smalltalk readings. 90
5.14 Safe/unsafe uses of Smalltalk writings. 90
5.15 Safe/unsafe uses of Smalltalk aliasing. 91
5.16 Per-feature distribution of user intention. 98

6.1 Examples of polymorphic type predicates. 123
6.2 Presence of type predicates in LOC, methods, classes and projects. 129
6.3 Predicates distribution based on constancy. 137

6.4 Refined constancy distribution, depending on predicate name. 138
6.5 Refining constant and variable predicates with the dynamic analysis. 141

7.1 A common structural protocol. 158
7.2 A common structural protocol across projects. 162
7.3 Definition of the instance relation on types. 166
7.4 Definition of the class relation on types 167

Chapter 1

Introduction

In the software development process, software engineers are always looking for reliable methods for module verification at several levels, *e.g.* verifying that modules behave correctly with respect to some specifications. Among all the methods in the literature, such as Hoare logic, denotational semantics, run-time monitoring, unit testing and others, *type systems* have become arguably the best established, the most widely used in practice, and the most lightweight formal method.

A type system is a tractable method for proving the absence of certain program behaviors [Pierce, 2002]. Therefore, a type system classifies program expressions according to the kinds of values they compute. These kinds of values are called *types*. A type is any property of a program expression that can be established prior to its evaluation. In other words, as a first approximation, a type represents an abstraction of a set of values. For example the type `Car` contains all cars in the system, but a bike is not a car, thus a bike does not have type `Car`. Types are the atomic elements of type systems. There are two main kinds of type systems that define how a given program is checked. They are called static type systems, where the set of values are restricted to an upper bound, and dynamic type systems, where the set of values is unrestricted [Cardelli, 1997a]. Each one has its own particularities, advantages and drawbacks.

"Dynamic" software development. In the software development community it is well known that dynamically typed languages, *e.g.* Php, Javascript, Python, etc., allow programmers to perform certain development tasks faster, such as prototypes or proofs of concept. An example is the great variety of web frameworks that

go from specific to general purpose. Usually those lightweight programs grow indiscriminately in order to support more features or to become part of a bigger (dynamically-typed) program. Unfortunately the more those programs grow, the more error prone they become.

With the continuous growth of programs written in dynamically-typed languages, the development process can end up being difficult, because dynamically-typed languages can not detect or emit any warnings on the presence of an error until this error occurs during the execution. This results in a high increase of debugging time, which negatively affects programmers morale. However there are several techniques, *e.g.* test-driven development [Beck, 2002] and code review, which decrease the error proneness of the development process. In a test-driven development process, programmers must first write test cases for each improvement or new functionality that they add to the system. In code review, a second programmer reviews the new code to find any issue. Those techniques can catch the most common (type) errors in the system, although they do not guarantee a one hundred percent type error coverage.

Introducing static types. Traditional static type systems can lessen the error proneness of programs written in dynamically-typed languages. After all,"well-typed programs cannot go wrong" [Milner, 1978]. In fact, with a static type system and appropriate type annotations, all type errors can be caught at compile time. Therefore, debugging time is considerably reduced, and with some compiler optimizations (which are possible because static information about the code that will be executed is available) the execution time can be improved. But the use of static type systems incurs some cost on the development process. Therefore the introduction of types in a dynamically typed language requires a careful analysis and study. These new kinds of type systems, specially targeted to existing dynamically-typed languages, are called retrofitted type systems.

Problem statement and thesis in a nutshell

Problem statement:

Smalltalk is a dynamically typed, object oriented, highly dynamic language. There are several Smalltalk dialects; the best-known being Squeak, a platform

for developing experimental educational software. However, there are dialects targeted for industrial and professional software, such as VisualWorks and Pharo. Those Smalltalk dialects are in continuous development, and have communities and users around them. Hence the appeal of introducing static types to Smalltalk is attractive to them. For instance, frameworks, such as the popular web framework Seaside, can be more reliable with the introduction of types. However, designing and implementing such a type system is a challenging problem because:

- Designing and implementing a retrofitted type system is a challenging task per se. Designing and implementing such type systems usually require providing typing support for most language features, *e.g.* metaclasses, and programming idioms, *e.g.* manual type-based dispatch. Supporting language features is particularly important in providing useful type system feedback at compile time, *e.g.* which method is wrongly called. Supporting programmer idioms is important for backward compatibility (*e.g.* maintaining support for the legacy packages) and seamless integration (*e.g.* programmers should not have to refactor code). To achieve such a type system, first we have to define a partial base type system (*i.e.* a core type system that combines static and dynamic typing) that best suits the language, and then propose a set of typing extensions that cover most language features and idioms. However adding typing extensions to the base type system can significantly increase its complexity. Consequently, each feature we add needs to be properly justified with supporting evidence. Therefore, the design and implementation of retrofitted type systems is complex.

- Smalltalk particularities complicate the design and implementation of a type system, or even any static analysis tool. Smalltalk is a dynamically typed, object oriented and highly dynamic language. In Smalltalk, everything is an object, even classes and programs. Smalltalk has a strong reflection API that allows programmers to alter any aspect of their programs at runtime. Programmers create Smalltalk code in a "live" environment, hence they can rapidly get feedback about their code. Such features exacerbate the challenges of designing and implementing a retrofitted type system.

Thesis:

> *Empirical evidence improves the design and implementation of retrofitted type systems whose main concern is supporting mostlanguage features and idioms*

In this dissertation we present Gradualtalk[1], a retrofitted type system for Smalltalk that successfully meets the above challenges. In other words, we design and implement a practical type system for Smalltalk that covers most (if not all) Smalltalk features and idioms. We propose a novel type system design that is composed of three parts: understanding Smalltalk features and idioms; defining the best partial type system; and carefully discussing and justifying with empirical evidence a set of suitable typing extensions. This will include two large-scale and three preliminary empirical studies that help us to make informed decisions. We then present the implemented type system and its validation through typing several Smalltalk projects. The type system of Gradualtalk combines several state-of-the-art features, such as gradual typing, unified nominal and structural subtyping, self type constructors for metaclasses, and incremental type checking. Gradualtalk is designed to ease the migration of existing, untyped Smalltalk code to typed Gradualtalk code.

Organization of the thesis

The remainder of the thesis is organized as follows. We first start with a chapter that presents the preliminary background related to type systems, followed by two parts that address the main developments of this work: an empirically-driven design of a retrofitted type system, and its implementation and validation. Finally, we conclude in a third part by summarizing the contributions of our work and highlighting potential future work.

Preliminaries

Chapter 2 presents a preliminary background (mostly focused on type systems) required in this thesis. However, additional background, such as that related to

[1] Available at http://www.pleiad.cl/gradualtalk

empirical studies, is introduced later. In particular this chapter presents a brief introduction to the different kinds of type systems in the literature and their presence in Smalltalk.

Part I: Empirically-Driven Design

The first part details the design of Gradualtalk (a retrofitted type system for Smalltalk), as well as the performed empirical studies to make informed decisions. In particular:

Chapter 3 presents the design of Gradualtalk. The design is divided into three parts: An introduction of Smalltalk features and programming idioms relevant for a type system. The definition of the type system goals (*i.e.* the primary intention to introduce the type system) that guide the development and the selection of the base type system (this is the particular partial type system to implement, *e.g.* gradual typing). Finally, the typing features that extend the base type system.

Chapter 4 reports on the use of `self` as a return value, the use of variables and selectors that may have different representations, and the use of collections. These three preliminary empirical studies help us make informed decisions regarding self types, union types and generics in Gradualtalk.

Chapter 5 reports on the use of dynamic and reflective features of Smalltalk. This empirical study helps us make informed decisions regarding the relevance of Smalltalk dynamic features, such as reflection.

Chapter 6 reports on the use of type-based dispatch patterns in Smalltalk. This empirical study helps us make informed decisions regarding this programming idiom.

Part II: Gradualtalk

The second part describes the implemented gradual type system by giving both an introduction of it and its typing extensions, and presenting an early valida-

tion of Gradualtalk through the typing of several Smalltalk projects. In particular:

Chapter 7 gives an introduction of Gradualtalk. Specifically, each feature of Gradualtalk is shortly defined and presented using code snippets.

Chapter 8 presents an early validation of Gradualtalk by the typing of seven real-world Smalltalk projects. Additionally, this chapter presents the limitations of Gradualtalk and the challenges of typing Smalltalk code.

Part III: Conclusions

In the final part of the thesis, Chapter 9 summarizes the contributions of this work, and Chapter 10 discusses potential directions for future work.

Related Publications and Implementations

The empirical studies presented in Part I were published by Callaú *et al.* [Callaú et al., 2013; Callaú et al., 2014]. Gradualtalk in Part II was published by Allende *et al.* [Allende et al., 2014a] in collaboration with the author and other researchers. Tangentially related to this thesis, the author also co-authored an implementation of a dependency tracking system, called Ghosts [Callaú and Tanter, 2013], which is the basis of the incremental type checker in Gradualtalk.

The implementation of Gradualtalk is available at http://pleiad.cl/gradualtalk. Additionally, the static and dynamic analyzers used in chapters 4, 5 and 6 are available at http://ss3.gemstone.com/ss/SimpleInspector, http://www.squeaksource.com/ff, and http://ss3.gemstone.com/ss/TOC/ respectively. Ghosts is available at http://pleiad.cl/ghosts.

Chapter 2

Type Systems and Smalltalk

In this chapter, we present the preliminary concepts and the state-of-the-art themes surrounding this work. This background is not exhaustive and some specific concepts are presented later in their respective sections.

The first section introduces the necessary type system concepts: Section 2.1 presents static type systems; Section 2.2 introduces dynamic type systems; Section 2.3 reviews most important state-of-the-art partial type systems; and Section 2.4 presents the concept of retrofitted type systems and reviews the most significant examples in the literature. Section 2.5 reviews several attempts to introduce types into Smalltalk. At the end, Section 2.6 revisits the problem statement. Readers already familiar with any of these topics may safely skip the corresponding sections or the entire chapter.

2.1 Static type systems

A static type system can be regarded as one that calculates a static approximation of the runtime behavior of all the terms in a program. Languages such as Java, C, ML and Haskell use this kind of type system.

Statically-typed languages define and enforce types at compile time. Those languages are usually explicitly typed, because types are part of their syntax,

e.g. Java and C. Other languages such as ML and Haskell include an inference engine that assigns types to well-formed expressions, making type annotations optional.

The benefits of static type systems are several and well-known. Some of them are: static type systems represent a very valuable first line of defense against programming errors, because they can catch type errors early in the development cycle; static type systems use type information to verify the absence of some bad program behaviors, *e.g.* invocation of a method that is not implemented in the receiver object; in statically typed languages, efficiency improvements can be obtained by eliminating many of the dynamic checks that would be needed to guarantee a type-safe execution [Pierce, 2002]. Although there are benefits, static type systems have a number of drawbacks [Tratt, 2009]:

- Static type systems are usually not flexible and any change in the software requirements can mandate a whole re-factoring. Static type systems often prevent transparent software evolution, because they require that the system as a whole is always type correct: it is not possible to temporarily and selectively turn off static type-checking.

- Static type systems are too conservative; in other words, certain valid programs will be rejected by the type checker.

- Relatively small increases in the expressivity of static type systems cause a disproportionately large increase in language complexity.

The above disadvantages can be solved (partially in some cases) by using a dynamic type system (see Section 2.2) or by deferring part of the checking to runtime. For instance, several languages—such as Java—support explicit type coercions (called *casts*), whose effect is to defer a type check to runtime. For instance, in Java, `Point p = (Point) collection.get(1)` gets the first element of a collection (of declared type `Object`), and casts it to `Point`. At runtime, a check is performed in order to verify that the returned object is of type `Point`. If not, a runtime exception is thrown. Furthermore, parametric polymorphism (aka. generics), introduced in Java 1.5, helps to avoid casts in many scenarios like the above, but at the cost of increased language complexity.

2.2 Dynamic type systems

Dynamic typing, also referred to as "dynamic checking" [Cardelli, 1997a], differs from static typing in the stages at which types are enforced; in dynamic type systems, type checks are deferred until runtime. Dynamically-typed languages use runtime tags to distinguish different kinds of structures. Actually most dynamically-typed languages guarantee safe execution [Krishnamurthi, 2007]. A type-less language, such as assembler, cannot guarantee safe execution; also, some statically-typed languages, like C, are unsafe because their runtime system does not uphold the guarantees put up by the static type system.

There are a lot of languages that are dynamically typed, *e.g.* Smalltalk, Python, JavaScript and others. They often are used in the industry to provide agile support to systems and fast adaptation to changing requirements. Features like those make dynamically-typed languages desirable in modern software development. In general, dynamically-typed languages allow for writing code that is more expressive than what would be possible with typical statically-typed languages. However dynamic type systems also have some drawbacks [Tratt, 2009]:

- In practice, a program written in a dynamically-typed language is slower than its equivalent written in a statically-typed language. This is because dynamic type systems inject runtime checks that are executed every time the program runs, whereas static type systems usually assist compilers in producing more optimized machine code.
- Dynamic type systems cannot detect errors or emit warnings prior to the execution of the program. Usually, those errors manifest in a different place than the one where the original error occur. Such errors may be difficult to locate. Consequently, dynamically-typed programs are inherently more error prone than statically-typed programs. As a way to mitigate this issue, developers have to focus on additional validation methods, such as unit testing, to detect most type errors.
- Statically-typed programs embed an implicit form of documentation. With a dynamically-typed language it is possible to informally annotate the expected

types of a function in comments, but these neither represent any guarantee nor are (usually) updated if the function specification changes.

- Some features of sophisticated Integrated Development Environments (IDEs), such as code completion, are enabled by type information; such features cannot be provided easily for dynamically-typed languages.

Static and dynamic typing seem to be antagonistic to one another, but they are actually complementary. This is because some advantages of one are disadvantages in the other, *e.g.* documenting APIs and agile development. So the following natural questions arise: Is it possible to integrate both? If so, how can that integration assist programmers in the software development process?

2.3 Partial type systems

The main difference between statically-typed and dynamically-typed languages is the stage at which types are enforced. In statically-typed languages types are enforced at compile time, while in dynamically-typed languages types are checked at runtime. This subtle difference results in a series of advantages and drawbacks described in previous sections. Beside these differences, we can see static and dynamic type systems as complementary: the advantages of one are the drawbacks of the other. Integrating both is the next logical step in a process that has been studied for a long time. From the *dynamic type* of Abadi *et al.* [Abadi et al., 1991] to most modern partial typing techniques, *i.e.* those that combine static and dynamic typing such as *gradual typing* [Siek and Taha, 2006], *like types* [Wrigstad et al., 2009] and others, researchers have been studying this particular symbiosis.

There are numerous ways to combine static and dynamic typing. For example, some dynamically-typed languages, such as Common Lisp, have optional type annotations that are used to improve runtime performance, but not to increase the amount of static checking. Another example is when statically-typed languages add a "dynamic" type in order to increase flexibility, but such languages require programmers to manually insert coercions to and from the "dynamic" type. Others allow programmers to annotate code with types, but enforcing them is optional;

they are called optional type systems [Bracha, 2004]. In this subsection, we present the most relevant partial typing techniques related to this thesis.

Soft-typing. Fagan and Cartwright [Cartwright and Fagan, 1991; Fagan, 1990] present a unification of static and dynamic type system, called Soft-typing. Later, Wright and Cartwright propose a practical implementation on Scheme [Wright and Cartwright, 1997]. A soft-type system infers types (informally called soft-types) for each component in the system, *e.g.* functions and data structures. Similarly to traditional type system, soft-types express program invariants. In the presence of ill-typed programs, a soft-type system should insert runtime checks. A soft-type system uses inferred soft-types to eliminate *unnecessary* runtime type checks, leaving only those that correspond to potential program errors. However, in particular cases some program expressions that could be determined to be erroneous are not properly considered as potential errors by the soft-type system. A soft-type system never rejects a program. Soft-type systems are recognized to be complex and brittle, *e.g.* inferring the type of a simple expression can produce an overly large type [Tobin-Hochstadt, 2010].

Optional and pluggable types. Bracha [Bracha, 2004] proposes that instead of a mandatory type system, programmers could choose from a variety of type systems for each module, *i.e.* different modules can be type checked by different type systems to catch specific problems. As a result, programmers can detect and prevent particular errors in problematic modules at compile time that would otherwise be detected at runtime. These *optional* type systems should neither require mandatory syntactic annotations in the program source, nor affect the runtime semantics of the language. Haldiman *et al.* [Haldiman et al., 2009] present a practical approach to pluggable types implemented in the Squeak dialect of Smalltalk [Black et al., 2007] where only pieces of code that contain partial type annotations are type checked using type inference and traditional static type checking. There are various implementations of pluggable types [Andreae et al., 2006; Dietl et al., 2011; Ekman and Hedin, 2007; Papi et al., 2008], but their main weakness is their lack of formalization. Optional typing has been widely accepted by several modern languages, *e.g.* ActionScript [Chang et al., 2007], PHP [Facebook, 2014],

JavaScript [Lerner et al., 2013], Dart [Bracha, 2011], Python [van Rossum, 2004], and Ruby [Furr, 2009]. The main advantage of optional typing is its transparency to the language semantic.

Gradual typing. Gradual typing [Siek et al., 2009; Siek and Taha, 2006, 2007] provides the advantages of both static and dynamic type systems. A programmer can control which part of the program must be statically type checked, and which part must be dynamically type checked. Typed code is statically verified; in contrast, untyped code is verified at run time (maintaining type flexibility). In a gradually-typed system, the notion of soundness means that the whole language is safe, and that a program either evaluates to a value or to a bad cast error. Cast errors can occur at the boundaries between the dynamic and static parts of a program. The integration of typed and untyped code is achieved through a widespread use of casts. Siek *et al.* [Siek et al., 2009] explored the design space of gradual typing (which depends on how and when casts are performed). They range from lazy to eager error detection strategy of high-order (*i.e.* function) casts: in a lazy strategy, high-order casts never fail immediately, they are checked only when the argument is applied; in a partially-eager strategy, high-order casts are checked immediately unless the source type (from which it is coerced) is the dynamic type; and in an eager strategy, high-order casts always perform some checking immediately. These strategies lead to an ample range of possible gradual typing implementations. In the past years, gradual typing has benefited from a series of extensions and implementations: Siek and Vachharajani [Siek and Vachharajani, 2008] propose a unification based type inference algorithm for gradual typing; Wolff *et al.* [Wolff et al., 2010] present a formalization of gradual typestate for Featherweight Java; Hernan *et al.* [Herman et al., 2010] research the space efficiency of coercions in gradual typing; Takikawa *et al.* [Takikawa et al., 2012] present a formalization of classes as first-class entities in gradual typing with mixin-based OO composition support; Ina and Igarashi [Ina and Igarashi, 2011] extend gradual typing with generics, and present an initial implementation on top of Java; Rastogi *et al.* [Rastogi et al., 2012] introduce gradual typing with optimized type inference support to ActionScript; recently, Vitousek *et al.* [Vitousek et al., 2014] present

Reticulated Python, a framework for the design of several gradual typing dialects in Python 3.

Hybrid typing. Hybrid typing [Flanagan, 2006] combines standard static typing with refinement types[1], which can be arbitrary predicates. An automated theorem prover is used to check the system consistency, and runtime checks are inserted where inconsistencies are detected. Similarly to gradual typing, hybrid typing rejects only programs that are *clearly* ill-typed; well-typed programs are accepted as is and those that are potentially not ill-typed are accepted after the addition of the necessary dynamic casts. A hybrid typing implementation has been presented by Gronski *et al.* [Gronski et al., 2006], who developed a practical hybrid typing system centered on precise interface specifications. However, there is no other concrete implementation of hybrid typing in any industrial language that evidence the benefits of hybrid typing, and there is no published work comparing hybrid typing with other partial typing systems.

Like types. Wrigstad *et al.* [Wrigstad et al., 2009] explore the evolution of untyped code to typed code. To that effect they present *like types*, a partial type system for a core object calculus (a scripting language called Thorn[2]). The like types system follows the traditional idea of partial typing, where the dynamic type (called dyn) coexists with static types (called concrete types in the work of Wrigstad *et al.*). Like types are intermediate types between dynamic and concrete types. For instance, declaring a variable of type like Point guarantees that the variable can only be used as a Point, without actually ensuring that the variable refers to a value of type Point at runtime. Like types are therefore only used to perform local type checking; in that regard, they are simpler and less powerful than other partial type systems, such as gradual typing. Thorn uses type information of concrete and like types to perform compiler optimizations speeding up the execution of scripts. Like types is a partial type system that does not include direct casts from the dynamic type to concrete types, but includes other implicit casts (those

[1] Refinement types [Freeman and Pfenning, 1991] are used to specify subsets of types (offering more precise type information). For example, the refinement type Integer $\wedge$ [5, 20] represents Integer values between 5 and 20.

[2] http://www.thorn-lang.org

that are to dynamic and like types). Currently there is no concrete comparison between like types and other partial type systems that widely use casts, like gradual typing.

2.4 Retrofitted type systems for dynamic languages

Retrofitted type systems are static or partial type systems designed to fit existing dynamically typed languages. One of the main goals of a retrofitted type system is to statically cover most untyped scenarios in the host language without forcing programs to be refactored. A retrofitted type system seems to be more appealing and practical than proposing a new typed language. This is particularly true when existing code has to be supported. As a common practice, type system designers start by identifying the set of runtime errors and proposing a typing feature to eliminate them statically. However, this technique is far from being optimal or satisfactory, *e.g.* languages with few runtime errors are hard to model, as reported by Lerner *et al.* [Lerner et al., 2013]. Additionally, not all runtime errors are actually important or practical to statically cover from the point of view of programmers, *e.g.* handling null references, because of theirs pervasive use. If a retrofitted type system tries to cover all possible untyped scenarios, it will end up with a typed language that will not fit existing code. Hence, designing a retrofitted type system is different and usually more complex than designing a type system for a new language.

Designing a retrofitted type system differs from designing a type system for a new language in the sense that more constraints must be taken into account by type system designers:

- Introducing types may impose structural restrictions to which existing code cannot comply. In this regard, existing valid code can be rejected by the type system.

  The most notable case is the simple typed lambda calculus, which can be considered as the first retrofitted type system. In the untyped lambda calculus, a turing-complete language, recursive functions can be defined in terms

of the fixed-point combinator, which is based on the omega combinator:

$$omega = (\lambda x.\ x\ x)(\lambda x.\ x\ x)$$
$$fix = \lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))$$

One particularity of the omega combinator is that it creates infinite loops when it is applied. When introducing types, we realize that omega cannot be properly typed, and therefore the simple typed lambda calculus becomes a non turing-complete language. As a solution, a new primitive must be added to the language [Pierce, 2002], and therefore all previously recursive code must be refactored accordingly.

- The presence of awkward (and particular) programming idioms (common code patterns) may overly-complicate the type system, or even make it undecidable. Programmers may be forced to abandon convenient programming idioms, affecting productivity[1].

  As example, consider the Smalltalk language where programmers can use symbols (constant string values) as replacement for block closures, *i.e.* lambdas. This particular programming idiom allows programmers to write code like:

  ```
  list collect: [:elem | elem printString].
  list collect: #printString
  ```

  Both lines produce the same results. Statically covering this scenario is complex for a static type system. Just a simple union type will not be sufficient because the symbol (which can be undecidable statically) must be a valid selector of the elements of the list.

- Some retrofitted type systems may impose changes in the language semantics, *e.g.* runtime casts. These changes can affect expected errors or just degrade the performance.

[1] Although productivity is hard to measure, we believe that preventing convenient idioms will force programmers to write more code.

For example, Allende *et al.* [Allende et al., 2013] report on how several cast insertion strategies affect the performance and modularity of retrofitted type systems. Another scenario has been reported by Tobin-Hochstadt [Tobin-Hochstadt, 2010] in the development of Typed Racket, where module level granularity between untyped and typed code reduces the performance impact compared to a more fine-grained granularity.

- Typing, or just re-compiling existing libraries may be unpractical, hence existing runnable code must co-exist with (new) typed code.

  For example, consider that existing compiled code does not carry any information about types. TypedRacket [Tobin-Hochstadt and St-Amour, 2012] solves this issue by introducing the `require/typed` form, which allows programmers to annotate types for functions and data structures from untyped modules, when importing them into typed modules. Another scenario is that of cast insertions, and whether casts must be inserted at method call sites or at the beginning of method declarations [Allende et al., 2013]. Inserting at method call sites may not be practical in pre-existing code, *e.g.* when recompilation is not possible.

In the literature there has been plenty of research on migrating dynamically-typed languages to their typed equivalent. However, in this introductory chapter, we only mention the most representative retrofitted type systems available today.

Typed Racket. Typed Racket [Tobin-Hochstadt and St-Amour, 2012] is an extension to Racket in order to support statically typed Racket programs. Typed Racket provides smooth and sound interoperability with untyped Racket [Tobin-Hochstadt, 2010], by using contracts at the boundaries. Typed Racket is one of the first to present a type system that is flexible enough to support existing programmer idioms. In fact, one of the philosophies of Typed Racket is that no new idioms need to be added because of the presence of types. This imposes the challenge of type checking most idioms in Typed Racket. A common idiom in (Typed) Racket is the use of type predicates, *e.g.* `string?`, to disambiguate the type of an expression. For example:

```
(: flexible–length ((U String (Listof Any)) → Integer))
 (define (flexible–length str–or–lst)
  (if (string? str–or–lst)
      (string–length str–or–lst)
      (length str–or–lst) ))
```

The function flexible–length computes the length of either a string or a list. A traditional type system cannot properly type this function, however, in Type Racket with Occurrence typing, the variable str–or–lst has type string in the true branch and type list in the false branch. Occurrence typing assigns differing types (with more precise information) to different occurrences of variables based on both type predicates and the control flow of the program. In the above code, Occurrence typing allows Typed Racket to successfully type check it, and even to precisely identify if str–or–list is either a string or a list depending on which branch is used.

Typed Racket includes several features, such as union types, occurrence typing, first-class polymorphic functions and local type inference. It is designed for the functional core of the Racket language. In Typed Racket, the granularity of typed and untyped code boundary is at the module level: a whole module is either entirely typed or not at all. Expression-level boundaries are more costly but more flexible, in that it is possible to statically type portions of a class while leaving a few difficult expressions typed dynamically. According to this design philosophy, Typed Racket does not support explicit type casts. The limitations of the per-module approach have been reported by Figueroa *et al.* in an experiment to implement a monadic aspect weaver in Typed Racket; they had to resort to the top type Any everywhere in their system [Figueroa et al., 2012].

DRuby. DRuby [Furr, 2009] is an optional static type system for Ruby, which uses type inference. Programmers can annotate their code, such as methods, and DRuby will check these annotations using runtime contracts on suspicious code, *i.e.* DRuby infers type to discard ill-typed code. DRuby is a retrofitted type system, hence it is devoted to supporting programming idioms. An idiom in Ruby, and other dynamically typed languages, is the use of a variable with different kinds of values. For example:

```
 foo = 42
 ...
```

```
foo = "string text"
foo.length
```

In the above code foo starts with a numeric value and it is used to perform some operations, and a string value is later assigned to foo and used again. DRuby does not reject the last two lines, because DRuby can properly infer the types of foo. This is because DRuby performs a flow-sensitive analysis for variables.

DRuby includes union and intersection types, structural types (called object types), parametric polymorphism and self types, among others. Furthermore, DRuby introduces a novel dynamic analysis to infer types in highly dynamic language constructors, *i.e.* the use of eval, send and missing_method functions. While Ruby has proper classes as objects and class methods, DRuby does not support the notions of typed meta-classes. This means that constructor (class) methods and uses of class cannot be precisely typed. Finally, DRuby is not a partial type system, so not all Ruby programs are valid DRuby programs.

TeJaS. TeJaS [Lerner et al., 2013] is a parametrized framework for building retrofitted type systems for JavaScript. Instead of developing a whole (and fixed) type system, TeJaS allows programmers to construct particular type systems by extending a base typed language. The base type system tries to cover most JavaScript particularities. One case is the extensive use of Strings as first-class entities, *e.g.* as field names in objects. This particularity of JavaScript is a challenge for any static type system. TeJaS proposes to replace Strings by regular expressions, therefore string values can be expressed exactly, and even families of strings can be expressed, for example:

```
{ ("get".+)  : → Int , ("set".+)  : Int → Void }
```

The above code expresses a type that represents Bean objects with int-typed parameters.

The base typed language represents the essence of typing JavaScript, therefore it is quite large and complex. However, this base type system has been modularized in seven layers to simplify its usage. Programmers can extend the base type system by modifying a single layer, *e.g.* its type environment can be modified to model various execution contexts of JavaScript without affecting any other

layer. The base type system supports local type inference to reduce type annotations, and syntactic sugar can be added to support most common patterns of type constructions, *i.e.* programming idioms. Type annotations are inserted through JavaScript comments, which constrain this type system to be optional.

Other type systems. In addition, type systems have been developed for other dynamic languages, *e.g.* Python, JavaScript and ActionScript. RPython (Restricted Python) [Ancona et al., 2007] is a statically-typed *subset* of Python, in which some dynamic features (*e.g.* dynamic modifications of classes and methods) have been removed. JS_0 [Anderson et al., 2005] is a statically-typed version of JavaScript with inference, where both dynamic addition of fields and method updating are supported. ActionScript is one of the first languages used in the industry to embrace gradual typing, and efforts have been made to optimize it using local type inference [Rastogi et al., 2012].

2.5 Type systems for Smalltalk

Many type systems have been proposed since the creation of Smalltalk-80. Most of them are focused on (traditional) static typing. In this section, we present the most relevant typing efforts in Smalltalk related to this dissertation.

Strongtalk and Pegon. Strongtalk [Bracha and Griswold, 1993] is a well-known statically typed Smalltalk dialect that incorporates several typing features. The Strongtalk type system is optional: it does not guarantee that the assumptions made in statically typed code are respected at runtime. There are two major versions of Strongtalk. The first one relies on a structural type system using brands (named structural protocols). The second version abandons brands and uses declared relations to determine subtyping. The main reason reported by Bracha for this change is the fact that structural types do not appropriately express the intent of the programmer, and are difficult to read, especially when debugging [Bracha, 1997]. Strongtalk is a full-featured static type system for Smalltalk. Among its typing features, the most remarkable ones are parametric polymorphism, and self types, including first-class classes support with some minor limitations. Finally,

Bracha states *"Strongtalk is not designed to type Smalltalk code without modifications"* [Bracha and Griswold, 1993]. In other words, Strongtalk is not meant to be a retrofitted type system for Smalltalk, however Strongtalk code can be easily converted to Smalltalk code by removing type annotations.

Pegon [Smit, 2012] is a recent optional type system for Smalltalk, inspired by Strongtalk. It includes all typing features and limitations of Strongtalk, and adds type inference, subtraction types and explicit casting. Pegon is still in development and authors reported that Pegon is only partially sound. Additionally, no report has been published yet about their experience with both the implementation and porting Smalltalk code. As a final remark, Pegon and Strongtalk are optionally-typed: they do not enforce any guarantees at runtime about the types of values [Bracha, 2004].

Other type systems for Smalltalk. The previous type systems are just a few examples of type systems for Smalltalk. There are several other proposals for Smalltalk [Graver and Johnson, 1990; Haldiman et al., 2009; Johnson, 1986; Johnson et al., 1988; Palsberg and Schwartzbach, 1990, 1991; Pluquet et al., 2009], although Strongtalk is the most representative and complete. Johnson and others [Graver and Johnson, 1990; Johnson, 1986; Johnson et al., 1988] present Typed Smalltalk, a static type system for Smalltalk with a strong focus on compiler optimizations. However Typed Smalltalk does not achieve the level of Strongtalk's optimizations. Palsberg *et al.* [Palsberg and Schwartzbach, 1990, 1991] present a type system for Smalltalk that does not rely on type annotations, instead it infers all possible types. This type system performs a global analysis, which is not convenient for incremental programming. Haldiman *et al.* [Haldiman et al., 2009] present a practical approach to pluggable types implemented in Smalltalk where only code that contains partial type annotations is type checked using type inference and traditional static type checking. Finally, Pluquet *et al.* [Pluquet et al., 2009], present RoelTyper, a type reconstruction tool for Smalltalk that infers possible nominal types for variables (instance, temporary and argument variables, as well as method returns) with an accuracy of 75% (tested on three Smalltalk applications).

2.6 Problem statement

Designing and implementing a practical retrofitted type system for a dynamically typed language is a difficult task because several factors must be taken into account. For instance, a naive introduction of types can reject programs that are valid. Refactoring could be a solution, but it is not practical and may be harmful for programmers to adopt the type system. As in any language, programmers rely on various programming idioms, some of which are challenging to type properly. Some typing features may introduce runtime coercions, which may affect expected errors and performance. Finally, existing modules must co-exist with new typed modules. Hence type system designers must carefully analyze which features are more appropriate to implement than others. Supporting all features could result in an undecidable or overly-complex type system. On the other hand, a permissive type system (such as a simple gradual type system) does not provide many benefits in terms of static verification. Therefore, designing a retrofitted type system requires significant extra work due to the difficulty in determining the most appropriate typing features.

The above challenges are even more exacerbated when we introduce types in Smalltalk. This is because of the highly-dynamic nature of the language and the "live" programming environment approach. Indeed, Smalltalk is a reflective language that in general, cannot be easily typed. A traditional static type system would require several complex features to fit Smalltalk, and therefore using type annotations will be cumbersome. Moreover, incremental programming in Smalltalk implies accepting partially-defined methods by the type system and to dynamically (at runtime) react to class updates. Additionally, as in any language, programmers rely on various particular Smalltalk programming idioms, some of which are challenging to type properly. These Smalltalk particularities make the design and implementation of a partial type system a challenge in itself.

This thesis presents the design and implementation of a pragmatic retrofitted type system for Smalltalk. Part I of this dissertation presents the type system design and two empirical studies performed on a large Smalltalk code base to

support part of the design decisions. Part II presents the proposed type system and its validation.

Part I: Empirically-Driven Design

Chapter 3

Designing A Retrofitted Type System

This chapter presents the design decisions for developing a retrofitted type system for Smalltalk. We first give a general introduction and explain our motivation for designing and implementing a type system for Smalltalk (Section 3.1). We then explore the Smalltalk language to better understand its particularities from a type system point of view (Section 3.2). Finally, we propose a partial type system (Section 3.3) and a list of suitable typing features to be implemented (Section 3.4).

3.1 Introduction

Retrofitted type systems are type systems specially designed to fit an existing dynamically-typed language. They can be statically typed or partially typed—where concrete types, such as Boolean, coexist with the dynamic type. One of their main goals is to statically cover most language features and common idioms. The development of a retrofitted type system is a challenging task, especially on the design side (see Section 2.4). On the one hand, supporting all possible typing features could produce an undecidable or overly-complex type system, or designers could even end up with a typed language that will not fit existing code. On the other hand, a permissive type system does not provide strong guarantees. Hence, the design of a type system must be based on some primary design goals. The

most recurrent design goal amongst existing retrofitted type systems is to achieve a high coverage, *i.e.* supporting both language features and programming idioms as much as possible. Hence, based on this type system goal, design decisions can be categorized in one of following groups: preventing certain runtime errors, covering specific programming idioms, or providing a significant benefit for programmers. However, they must be properly backed up by supporting evidence to achieve a pragmatic retrofitted type system.

Smalltalk [Goldberg and Robson, 1983] is the emblematic dynamic object-oriented language and has served as inspiration for many recent languages. Smalltalk is still used in industry, and the appeal of partial typing is attractive to many. In particular, the recent Pharo dialect of Smalltalk, targeted at robust industrial development, is in need of such a system. The dialect with the most developed static type system for Smalltalk, Strongtalk [Bracha and Griswold, 1993] is an optional type system [Bracha, 2004]. Optional type systems do not influence the runtime semantics (runtime checks are not inserted) of the language and therefore do not enforce any guarantee about the type of values at runtime. This is very different from other partial type systems, for instance, in a gradual type system which does ensure that assumptions made by statically-typed code are not violated. If they are, the faulty dynamic code is blamed accordingly [Wadler and Findler, 2009]. The key to these guarantees is the insertion of runtime casts at the static/dynamic boundaries [Siek and Taha, 2006]. Hence, a partial type system that makes strong guarantees for Smalltalk code is still missing. In particular designing and implementing a partial type system for Smalltalk is a challenging task because of the highly-dynamic nature of the language and the "live" programming environment approach.

In this chapter, we present the process that we followed for designing a retrofitted type system for Smalltalk. The goal of this process is to choose a base type system with a list of proposed typing features that would be a "good fit" for Smalltalk. However the development of a type system is an incremental process, and hence the presented analysis is just an initial step towards this path.

We applied three steps to propose an initial design. First, we explore the Smalltalk language, its core foundations, language features and common programming idioms. Such analysis allows us to undersand Smalltalk particularities and

limitations. We then present the type system goals that will guide the design and implementation of the type system. Based on these goals, we propose a base type system. Finally we propose a list of typing features that will cover any remaining language feature or idiom that the base type system cannot properly type. These features are type system extensions that will be implemented on top of the type system.

3.2 The Smalltalk language

The first step in the design is to examine the Smalltalk language to find relevant cases and scenarios for the type system. We are looking for:

- The core foundation of the language, which refers to the language foundation and values;
- Smalltalk representative features, *i.e.* the features of the language that define its capabilities and limitations;
- Most common programming idioms, *i.e.* formal or informal code patterns, reported by practitioners.

3.2.1 Smalltalk core

Smalltalk is a pure object oriented language, which means its values are only objects. In Smalltalk, everything is an object [Nierstrasz et al., 2010], even numbers and classes. Furthermore, control structures are the results of sending messages between objects. Each message contains a selector (method name) and arguments. When the object receives the message, it will look up the selector in its method dictionary, retrieve the associated method, and execute it with the arguments of the message (if any). Smalltalk's syntax is also distinct from C-like languages. We provide equivalent expressions for common cases in Table 3.1. These particularities and syntax make Smalltalk a simple and clean language.

Smalltalk	**Java**
foo bar.	foo.bar();
foo bar: baz.	foo.bar(baz);
foo bar: baz with: quux.	foo.bar(baz, quux);
p := Point new.	Point p = new Point();
↑ foo	return foo;
self	this
super	super
'String'	"String"
#symbol	String.intern("symbol");

Table 3.1: Smalltalk and Java syntax compared.

3.2.2 Smalltalk features

Smalltalk is a general purpose and industry-level programming language. This means it has several features that range from common and simple ones, like numbers, to complex ones, like reflection. In this part, we list most of them that are relevant for a retrofitted type system. For more details about these features, please see the "Pharo by Example" book series [Bergel et al., 2013; Nierstrasz et al., 2010].

Class-based objects.

```
"creating class Point"
Object subclass: #Point  instanceVariableNames: 'x y'

"defining a getter method"
Point>> getX
↑ x

"creating and using objects"
obj := Point new.
obj x.
```

In Smalltalk objects are class based, *i.e.* every object is an instance of a class. In the above example, obj is an instance of class Point. An object can access all methods defined by its class, *e.g.* getX. In contrast objects cannot directly access field members, such as x in the above example. Instead, programmers must provide getter methods, *e.g.* getX.

Classes as objects.

```
"creating a class by calling a method"
Point subclass: #Point3D  instanceVariableNames: 'z'
```

```
"defining a method in a metaclass"
Point3D class>>origin
↑ self new x: 0; y: 0; z: 0; yourself
```

Classes are first-class citizens and only support single inheritance. In fact, creating a class is just calling a method. In the example, class Point3D is created by sending a message to class Point, which is its super class. Classes are instances of their metaclasses, *e.g.* Point3D is instance of Point3D class, and hence metaclasses contain all available methods for their instances, like method origin for class Point3D.

Lambdas.

```
[:x | x + 1 ] value: 2      " ⟶ evaluates to 3"

numbers collect: [:num | num asString ]   " ⟶ evaluates to a list of strings"
```

Smalltalk supports first-class lambdas with lexical scope, called blocks. A block is an object, hence it has some specific methods defined in class BlockClosure. A block may take parameters, and it evaluates by calling method value or siblings. Blocks are particularly important in Smalltalk, *e.g.* control structures use them pervasively.

Control structures.

```
num > 500 ifTrue: [ 'big' ] ifFalse: [ 'small' ]

n := 1
[ n < 1000 ] whileTrue: [ n:= n*2 ]    " ⟶ n evaluates to 1024"

n := 1
10 timesRepeat: [ n:= n*2 ]            " ⟶ n evaluates to 1024"
```

Control structures, such as if or while, are expressed by sending messages. In the above examples, blocks are used to compute the final answer. In the case of conditionals to express the branches, and in the case of loops to express the block of code to be repeated.

Returning self.

```
MyClass>> helloWorld
 Transcript show: 'Hello world'
```

```
"↑ self      <–– this statement is implicit in the above method"
```

Methods without a return statement return self, *i.e.* the current object. This implies that the concept of void is not present in Smalltalk, and all methods always have a return value, which by default is the current object. This allows programmers to chain method calls easily.

Traits [Schärli et al., 2003].

```
"trait creation"
Trait named: #TSortable uses: {}

"a method trait definition"
TSortable>> sort
...

"another method trait definition"
TSortable>> isSorted
...
```

Traits are composable units of behaviors, similar to mixins [Bracha and Cook, 1990]. Programmers use traits in Smalltalk to define a set of common methods that a group of objects with different super classes implement. Traits allow programmers to simulate multiple inheritance in Smalltalk. In the example, trait TSortable is a trait that specifies and defines how collection objects are sorted. This trait is used by several classes in the Collections library.

Live programming.

Smalltalk is considered to be a live programming environment. This is because of the incremental and fragmented (*i.e.* in a method basis) program construction. Additionally, programs and objects are encapsulated inside a Smalltalk image. This allows objects to store their state, and programs are stored as objects too, *e.g.* methods are instances of class CompiledMethod. Such a rich environment allows programmers to get very rapid feedback while programming, *e.g.* when a class adds or removes instance variables, all subclasses are re-compiled and objects are updated accordingly.

Open classes.

```
Object>> asMorph
...

ClassDescription>> browserIcon
...
```

Programmers can freely add methods to any class in the system at any time. For example, method asMorph is a method added to the class Object by the Morph package (UI Smalltalk library). Another example is method browserIcon added to the class ClassDescription by the OmniBrowser package (a system browser tool). As mentioned in the previous feature, live programming, subclasses are recompiled after the method is added. This feature makes Smalltalk programs and APIs very extensible, but at the cost of overly populating some classes, such as Object.

Reflection.

```
Point allMethods    " ⟶ evaluates to the list of compiled methods of Point"

sum := #+ .
1 perform: sum withArguments: {2}    " ⟶ evaluates to 3"

"updating Point3D super class"
Point3D superclass: AnotherPoint
```

Smalltalk reflective capabilities allow programmers to both inspect and modify object structures and definitions at runtime. In the above code, we show three cases: inspecting all methods of a class, performing a method invocation, and updating the super class. Such reflective features make Smalltalk an agile and versatile programming environment; however, at the same time, they make implementing static tools for Smalltalk a challenge.

3.2.3 Programming idioms

Programming idioms are those formal or informal patterns that programmers commonly follow. We can find good examples in mainstream languages, for instance in Java with type-based dispatch patterns:

```
Object foo = new Foo(...);
...
if (foo instanceof Foo) {
  ((Foo) foo).doFooStuff();
```

```
}
```

In this scenario a type system can exploit the type information in the true branch, and consequently, programmers can avoid having to write casts.

Type system designers must focus on relevant programming idioms where a type system may get useful information to benefit programmers. Therefore, not all idioms are relevant, for instance in a Java-like language, getter/setter methods might be considered as a very common idiom, however this idiom may not be really relevant from a type system perspective. Hence, type system designers must carefully analyze which idioms are worthy of being covered in the type system.

In the case of Smalltalk, we find five idioms as the most relevant to be considered in a retrofitted type system. One person (the author of this thesis) proposes these idioms based on his experience as a Smalltalk programmer. Then, two others (advisors of this thesis) validate those idioms as the most relevant for the introduction of types in Smalltalk. Similar to the language features, we add a short name (in italics) to each idiom for later reference.

Abstract methods.

```
AbstractFoo >> abstractMethod
  self subclassResponsability
```

Smalltalk does not support abstract classes, but programmers can emulate them through calling the subclassResponsability method. Calling it at runtime causes an exception. This idiom is particularly important, because several classes in the kernel system use it pervasively, *e.g.* Behavior, Boolean and even Object.

Inappropriate methods.

```
Dictionary >> remove:
  "Dictionary is subclass of Collection, which implements #remove:.
  Dictionary provides #removeKey: instead."
  self shouldNotImplement
```

This idiom is due to the limitation of single inheritance, and it is usually considered a design fault. However, sometimes it is hard to avoid such workarounds. Similar to the previous idiom, this pattern is frequently used, especially by classes

in the Collections library, *e.g.* ArrayedCollection, ByteArray and Matrix.

Joining values.

```
strOrNum ifTrue: [ 'a string' ] ifFalse: [ 1 ]

AbstractFont>> widthOfStringOrText: aStringOrText
...        "aStringOrText can be an instance of String or Text"
```

This is a programming idiom that is present in several dynamically typed languages. In this idiom, programmers join two different values in a single variable or expression. In the examples, the first one is an expression that evaluates to a string or an integer, and the second one is a method whose argument may be an instance of classes String or Text (not related hierarchically).

Type-based dispatch.

```
programNode isVariable ifTrue: [programNode name]

AbstractFont>> widthOfStringOrText: aStringOrText
    aStringOrText ifNil: [↑0].
    ↑ aStringOrText isText
         ifTrue:[self approxWidthOfText: aStringOrText ]
         ifFalse:[self widthOfString: aStringOrText ]
```

This is a defensive programming idiom that verifies whether an object complies with a given condition, *e.g.* a type predicate, before performing a specific action safely. In the first example, programNode is checked to be a variable, before calling method name—only available in VariableNode instances. The second example shows the implementation of method widthOfStringOrText, where two type predicates are check, isNil (implicitly in ifNil:) and isText.

Symbols as methods.

```
figures do: #draw

''instead of:''
figures do: [:fig | fig draw]
```

Smalltalk allows programmers to use symbols (lexical strings) as pseudo first-class methods. In other words, instead of using a block closure and calling the desired

method (draw in the example), programmers can use a symbol that represents the method name to achieve the same goal.

3.3 Type system goals and foundations

In this step, we first determine the type system goals that will govern all the type system development. Second, based on those goals, we have to choose which kind of (partial) type system will be the base type system for Smalltalk.

3.3.1 Type system goals

The type system goals are the primary intention of designers to introduce the type system in a dynamically typed language, for instance, preventing certain kinds of errors such as division by zero, or assisting compiler optimizations. However, defining these goals can be ambiguous. Therefore we define the type system goals by answering the following questions:

> *What kind of errors does the proposed type system avoid?* This will determine how many language features will be covered, how many programming idioms will provide more useful typing information, and how refined the type information will be. A common example in object-oriented languages is to avoid bad object member accesses, *e.g.* calling a wrong method.
>
> *What guarantees does the proposed type system give?* This has to do with the soundness properties, and how appropriate the type system feedback from errors to programmers is. For instance, Featherweight Java [Igarashi et al., 2001], as formulated by Igarashi *et al.*, guarantees that a well-typed expression will reduce to a value or get stuck in a bad type coercion, *i.e.* a cast.
>
> *How flexible is the proposed type system?* The flexibility of the proposed type system to cover untyped code is a principal concern. Hence, in this question, we define up to which level untyped code will be covered by the proposed type system. One example is that of Type Racket, where untyped modules

can be used in typed modules if they are explicitly imported and specify their signatures, *i.e.* expected types. Another is allowing untyped code, but restricting certain languages and idioms that are known to be complex to type, *e.g.* the eval function in JavaScript.

These goals regulate the type system development. Hence a technical specification of a type system, *i.e.* its foundations and features, is based on them. The next steps are defining the type system foundations and listing its most relevant typing features.

Smalltalk type system goals. To better define the goals, we answer the above questions regarding errors to avoid, type system guarantees, and flexibility:

Errors to avoid. In Smalltalk, everything is achieved through sending messages between objects, hence the primary expected exception in Smalltalk programs is a wrong member access, *i.e.* MessageNotUnderstood exception (MNU). We want to avoid it with the introduction of a type system[1]. Moreover, Smalltalk is a safe language, which means that there will not be unsafe pointer accesses, as opposed to C. This capability simplifies the effort to guarantee type safety in Smalltalk.

Guarantees. The introduction of a type system in Smalltalk will guarantee at least that well-typed programs will not reach a MessageNotUnderstood exception.

Flexibility. The proposed type system will be a retrofitted one, therefore another major design goal is that it should accommodate existing programming idioms in order to allow for an easy and incremental migration from untyped to typed code. More pragmatically, the proposed type system will allow programmers to properly type Smalltalk features and idioms as much as possible. In other words, all (or almost all) features and idioms in Section 3.2 will be “typable” by the type system. If a feature or idiom cannot be

[1]Smalltalk language and APIs implement several runtime exceptions, however MNU is the main exception in the kernel language, and the others are only specific to certain scenarios, like Halt for inserting breakpoints.

tractable by the type system, its usages may remain untyped. This means that typed and untyped code must coexist.

3.3.2 Foundations

The foundations are the core nature of the type system, *i.e.* which particular (partial) type system it is: static [Pierce, 2002], optional [Bracha, 2004], gradual [Siek and Taha, 2006], soft [Cartwright and Fagan, 1991], hybrid [Knowles and Flanagan, 2010], among many others [Garcia et al., 2010; Haldiman et al., 2009; Wrigstad et al., 2009]. The core of Smalltalk and the goals of the type system determine the type system foundations.

Gradualtalk foundations. We define the Smalltalk type system foundations based on the goals from Section 3.3.1. In details, we compare each kind of type system against those goals to determine which is the best.

Type systems are frequently related to traditional static type systems. In this particular case, a static type system for Smalltalk will not allow programmers to easily and incrementally migrate untyped to typed code. A static type system catches type errors at compile time and provides the necessary guarantees at runtime. However untyped code would be rejected by a static type system. Therefore, a static type system for Smalltalk will not satisfy the flexibility goal. Other alternatives to static type systems are partial type systems (see Section 2.3).

In the following paragraphs, we discuss all partial type systems (from Section 2.3) against our goals. Note that those partial type systems that do not comply (those of optional and soft typing) have a title in plain text, while those that satisfy the type system goals but are non-selected (*hydrid and like types*) have a title in italics. Finally, the selected one (**gradual typing**) has its title in bold.

- Optional typing. An optional type system would catch type errors at compile time, and untyped code would not be rejected. Nevertheless type system guarantees are not enforced at runtime by optional type systems. This is because optional type systems do not affect the language semantics, *i.e.* runtime checks are not inserted.

- Soft typing. A soft type system does not reject clearly ill-typed programs, instead it both tries to infer the appropriate type, and inserts runtime checks. Hence, a soft type system does not catch type errors at compile time.

- *Hydrid types.* A hybrid type system would satisfy our goals. However, it requires an automated theorem prover to check annotated types. Compared to gradual typing, which uses simple typing rules, implementing hybrid types requires an extra effort, *e.g.* defining and checking refinement types.

- *Like types.* A like type system would satisfy our goals too. However, like types (*i.e.* a form of pseudo-concrete types) are only used to perform local type checking. Hence, they are simpler and less powerful than gradual types. Additionally, gradual typing has an ample design space due to its different cast semantics.

- **Gradual typing.** A gradual type system would comply with all type system goals, as well as others too (see hybrid types and like types above). However, we decide to choose gradual typing for several reasons that leverage gradual typing with respect to others. Gradual typing has a strong theoretical basis with simple typing rules and an ample design space. Additionally, it has been extended with several typing features, and implementation experiences are available in the literature. Finally, a gradual type system would enable an easy and seamless incremental migration from untyped to typed code in Smalltalk.

As previously mentioned, we choose gradual typing as the base type system for our retrofitted type system for Smalltalk. In particular, a gradual type system would catch MessageNotUnderstood exceptions at compile time, guaranteeing that well-typed programs would not raise that exception. However, the type system guarantees defined earlier have to be updated to reflect the insertion of gradual types. Hence, it is updated as follows: *if a well-typed program raises a runtime exception, it is because of a bad type coercion (*i.e. *inappropriate coercion from an untyped value to a typed one)*, because gradual typing introduces implicit casts for untyped/typed code boundaries. We call this gradual type system for Smalltalk,

Gradualtalk.

3.4 Type system features

There are a lot of typing features available in the literature that can be implemented in modern type systems, but implementing all of them is not feasible or practical. Hence, all typing features must be targeted to cover the existing language features and idioms or properly justified to benefit programmers. We applied the following methodology for proposing and selecting typing features:

- We propose a set of typing features that come from a direct **mapping of language features and programming idioms**, *i.e.* each language feature and idiom is at least covered by a typing feature. For example, in class-based object-oriented languages, nominal subtyping makes sense, because programmers usually relate classes with types.

- Although all available typing features in the literature have some theoretical and practical benefits, implementing all of them may not be feasible or practical. Hence, for each typing feature we define the level of empirical evidence required. Those levels are:

  No evidence: Some typing features may not require empirical evidence. This is because such features may be the most natural fit for the language, or they may imply a specific type system design. In our case, just one typing feature, *nominal subtyping*, does not require empirical evidence, because it is a design decision and a natural fit for class-based object oriented languages.

  Literature: For some typing features there is empirical evidence available in the literature that is useful for making informed decisions. We use that evidence when there is not need for a preliminary study, *e.g.* the empirical evidence (from one or several studies) is strong enough to justify the typing feature.

Preliminary studies: For some typing features that are not critical or may not require a significant implementation and/or usage effort. We propose the use of preliminary empirical studies. Furthermore, the evidence from the literature can be combined with preliminary studies to make stronger justifications.

Large studies: Some typing features may require a broad or large empirical study to better understand their relevance for retrofitted type systems. Additionally, such large studies may also inform practitioners and researchers in other areas, *e.g.* refactoring and IDE tools.

3.4.1 Features and idioms covered by the base type system

Just the introduction of gradual typing will cover some language features and idioms. This is the case with the features *lambdas* and *control structures* that will be covered for free, which means that they are intrinsically covered by gradual typing. This is because gradual typing has been formalized for functional and object oriented languages, hence lambdas and control structures (simple object messages) are covered by them respectively. Other cases are idioms *abstract methods* and *inappropriate methods*, which can be easily supported by simply modifying the type system to mark those methods (and their owner classes) in the type system dictionary when programmers are compiling them. Hence, the type system can appropriately react when an instance of those marked classes is trying to call an abstract or inappropriate method.

3.4.2 Features and idioms covered by a typing feature

Just the introduction of gradual typing (with some minor modifications) covers two language features—*lambdas* and *control structures*—and two idioms—*abstract methods* and *inappropriate methods*. Additionally, using the dynamic type would allow programmers to type check every complex program, however any useful type information would be lost in the process. Therefore, additional typing features are required to properly type those cases where type information is lost. We categorize these features into three groups: those that cover language features; those that

cover programming idioms; and those that cover type idioms—code patterns due to the presence of types.

If a language feature or idiom is not covered by the type system, there are two alternatives for programmers. First, they can refactor the problematic code to comply with the type system, however this may not be practical. The second choice is to rely on the dynamic type (*i.e.* leaving those parts untyped) to use the problematic feature or idiom. In that case, gradual typing inserts casts to guarantee safe coercions between typed and untyped code, but excessive casts may degrade performance.

Language features. Table 3.2 lists the proposed typing features that cover one or more Smalltalk features. The first column is the common feature name, while the second column details it. Finally, the last column lists what language features are directly covered by each feature. From this set of initial features, we analyze their relevance for being included in the type system.

Feature	**Details**	**Covering**
Nominal subtyping	Types are induced by classes, hence programmers can easily express and enforce design intent.	Class-based objects
Self types [Saito and Igarashi, 2009]	It allows programmers to properly handle the type of `self` (*i.e.* the current object, `this` in Java-like languages).	Returning self, Classes as objects*
Structural subtyping	It allows programmers to explicitly specify a set of methods that objects must implement. It can be used to type common protocols (*e.g.* handling properties) or traits.	Traits
Incremental checking	Incremental (type) checking allows programmers to incrementally (re-)type check classes, such as method by method.	Live programming, Open classes
Effect system [Pierce, 2005]	Effect systems are used to track effects across typing expressions and statements. It is useful for typing structural changes in classes at runtime.	Live programming, Open classes, Reflection

Table 3.2: Proposed typing features for Gradualtalk that cover one or more language features.
* Self types requires being extended to properly cover the interaction of self and metaclasses.

Self types, nominal and structural subtyping.
We identify that features *self types*, *nominal* and *structural subtyping* cover important language features *returning self* and (partially) *classes as objects*, *class-based objects*, and *traits* respectively that no other typing feature covers. If any of these typing features is not implemented, Smalltalk programs may not be properly typed. Hence, we include them in a first iteration of Gradualtalk. We justify our decisions for each feature in the following points:

Self types. With self types [Saito and Igarashi, 2009], the pseudo-variable self in Smalltalk has type Self, *i.e.* the type of the current class. This is particularly important for covering returning self and classes as objects features. In fact, in a preliminary empirical study (see Section 4.3), we find that 45.8% of methods in the analyzed corpus return self, either explicitly (7.7%) or by default (92.3%). Additionally, in Chapter 5 we performed an empirical analysis on the prevalence of dynamic features, where one of the tracked features is the use of self (and other variables) to instantiate objects and create classes. Section 5.4 shows that self is used in 74% of instance creations, and Smalltalk programs do create classes at runtime. These results show the relevance of self types and the necessity to extend it to support metaclasses. Therefore, we include self types in Gradualtalk.

Nominal subtyping. With nominal subtyping, types are induced by classes, which helps programmers easily express and enforce their design intent. Actually, most modern object oriented languages (*e.g.* Java, C++, C# and Scala) are typed and class-based support nominal subtyping. Furthermore, class-based objects is a central Smalltalk feature. Hence supporting nominal subtyping is mandatory.

Structural subtyping. It offers the flexibility to describe the type of an object based on the set of methods that are allowed to be executed. This helps programmers specify the type of traits and common protocols, such as those that handle properties, in Smalltalk. Malayeri and Aldrich [Malayeri and Aldrich, 2009] study empirically the relevance of structural subtyping in Java. They found the prevalence of common implicit protocols, *e.g.* a

read-only non-iterable map, across several classes. In 19% of the methods there is a similar method (*i.e.* with the same name and signature) in another class with a different hierarchy. Additionally, they found that the introduction of structural subtyping can help programmers avoid the use of UnsupportedOperationException in Java, which can be comparable to the use of #shouldNotImplement in Smalltalk. Finally, Malayeri and Aldrich conclude that nominally typed programs could be improved with structural typing. Hence, supporting structural subtyping could be useful for Smalltalk programs beyond traits. Consequently, we include this feature in the type system.

Effect systems and incremental checking.
Effect systems [Pierce, 2005] and incremental (type) checking [Pierce, 2005] are typing extensions that cover language features that have overlap between them (see Table 3.2). In the following paragraphs, we introduce and analyze these extensions to make an informed decision of which should be included in the type system.

Effect systems. An effect system also known as a "type and effect system" is a typing extension that allows the type system to track all important effects of computation. An effect system would allow Gradualtalk to track structural changes in classes at runtime. Although this feature covers *live programming*, *open classes* and *reflection* language features, its implementation and usability cost (*e.g.* type annotations, theoretical complexity and undecidability) are too high for type system designers and programmers. This is because a type and effect system requires implementing a linear type system, and programmers may be forced to annotate their effects for each potentially dangerous method.

Incremental (type) checking. It is a typing extension that allows the type system to re-type check modules when one of their dependencies changed. This extension would track dependencies between classes and methods in Smalltalk and trigger automatic re-type checking. This feature does not require a significant implementation effort (a dependency tracking system is

only required) and it is transparent for programmers. However, this feature only covers *live programming* and *open classes*, leaving *reflection* uncovered.

If the reflection feature is not often used in Smalltalk programs or only used in specific kinds of programs, like tests, we can safely choose incremental type checking over effect systems. Therefore, a question arises: **How often are reflective features used in Smalltalk programs?** Chapter 5 helps us make informed decisions regarding the prevalence of dynamic features, especially reflective ones. Sections 5.4 and 5.4 show that reflective features are in general, rarely used, with the exception of message sending, whose usages are mostly unsafe (*i.e.* potentially undecidable) for a type system. Finally, Section 5.9 concludes that incremental type checking is more suitable to be implemented than an effect system, and hence Gradualtalk would support incremental type checking instead of effect systems.

Programming idioms. From all five programming idioms listed in Section 3.2.3, *abstract methods* and *inappropriate methods* are covered by gradual typing (see Section 3.4.1). This leaves three remaining idioms: *joining values*, *type-based dispatch* and *symbols as methods*.

Joining values. This idiom can be covered by introducing *union types*. This is because union types allow programmers to join several types in a single one. Using union types, the code examples presented in Section 3.2.3 can be typed by using the types String | Integer and String | Text:

```
strOrNum ifTrue: [ 'a string' ] ifFalse: [ 1 ]  "has type String | Integer"
```

```
AbstractFont>> widthOfStringOrText: (String | Text) aStringOrText
```

However, how prevalent is this idiom in Smalltalk? To answer this question, we perform a preliminary empirical study (see Section 4.4). We find that 1.4% of methods (1,035 out of 76,137) in the analyzed corpus use at least one of such variables or have a method name that may represent two different kinds of values. This shows that the idiom is potentially present. Furthermore extending the type system to include union types implies a low

effort, *e.g.* adding few new subtyping rules and computing structural intersection between types. Hence, we include this feature in the first iteration of Gradualtalk.

Type-based dispatch. The introduction of *flow-sensitive typing* [Guha et al., 2011] can properly type this programming idiom. This is because it allows the type system to properly re-type variables on control flow statements. However, the cost of implementing this feature may trump its benefits, if this idiom is rarely used or it is only used in specific scenarios. Therefore, a question arises: **How prevalent is this idiom in Smalltalk programs?** Chapter 6 helps us make informed decisions regarding the prevalence of typed-based dispatch patterns. This study shows that type predicate checks (the core of this idiom) are present in Smalltalk programs with a density of one for each 50 lines of code. However, more than three quarters (76%) are for checking object nullness. Hence a more specific typing approach, such as non-null types, would have more effect with significantly less effort. Nevertheless, we consider the frequency of use of these patterns too dispersed across applications to be considered as a priority idiom to cover. Additionally, the interaction between flow-sensitive typing and other features in Gradualtalk, *e.g.* gradual typing and generics, requires a deeper analysis. Therefore, in Section 5.9, we conclude that flow-sensitive typing will not be implemented in Gradualtalk for now.

Symbols as methods. This feature is very complex to statically type check. First, it uses symbols to specify the calling method, which in some scenarios cannot be statically determined, *e.g.* if it is derived from a user input. Second, if we get the calling method, it would be hard to compute the type of expression, especially if the receiver contains untyped elements (*i.e.* where its type is the dynamic type). This idiom can be compared to the reflective feature of message sending. In fact this feature uses `perform:` to call the method reflectively. Section 5.4 shows that message sending is the most common dynamic feature of Smalltalk (29.7% of all traced occurrences), but a huge percentage of them (93%) are unsafe, and hence hardly tractable by

a type system. Although these results cannot be directly related to this idiom, it shows the complexity of covering language features and idioms based on reflection. Therefore, we exclude this idiom in the first iteration of the design and implementation of Gradualtalk. As mentioned before, a direct consequence of not covering an idiom is that programmers have to refactor or leave these portions of code untyped.

Typing idioms. The presence of types forces programs to comply with a series of rules in order to be accepted by the type system. However in some scenarios, programmers may be forced to use programming idioms to satisfy the type system. We informally call them, *typing idioms.* We mention two typing idioms that are worthy of being covered:

Explicit coercions. Gradual typing introduces casts, but they are implicit, *i.e.* programmers are not allowed to explicitly use them. However explicit casts are necessary in some scenarios, *e.g.* specifying the return type of a untyped method, or updating the type of an expression to a more general (upcast) or specific type (downcast). Additionally, supporting explicit casts is simple to implement, since it just requires adding a new syntax constructor in the language. Because of these arguments, we let programmers use casts explicitly.

Collections. With the introduction of types, programmers are forced to store elements in collection objects blindly, which means the type information is discarded and accessing the element would imply manually casting it to safely use it:

```
|(Array) objs|
...
objs at:0 put: 'a text'.
 "inserted element is a string, but the array does not know it"
...
(<String> objs at: 0) capitalized  "accessing the element requires a cast"
```

In order to avoid similar scenarios as in the previous example, we propose to include generics (*aka.* parametric polymorphism) in Gradualtalk. Generics allow programmers to write function and data types (*i.e.* classes) generically,

while maintaining static properties. Despite these theoretical benefits, generics have been studied in practice with limited proven benefits. This is the case with the following studies, Parnin *et al.* [Parnin et al., 2013], and Hoppe and Hanenberg [Hoppe and Hanenberg, 2013]. Parnin *et al.* study the introduction of generics in 20 open source Java projects. This study shows that over half of the projects did not use generics, with most usages (25%) corresponding to List<String> or similar cases. However, results by Parnin *et al.* suggest that generics may help reduce casts and runtime errors with regard to code duplication, but in localized cases rather than in general. Hoppe and Hanenberg study whether generic types (*e.g.* List<String>) provide significant benefits compared to raw types (*e.g.* List). They find somewhat mixed results: generics improve API documentations, while extensibility is negatively affected, and there are no significant differences between raw and generic types regarding fixing type errors. Those studies suggest that introducing generics is somewhat controversial. Furthermore, in an interview, Gilad Bracha recognizes that generics is a controversial feature (especially its variance schemes), but still necessary [Tratt and Welc, 2014].

In order to better understand the possible impact of generics in Smalltalk, we perform a preliminary empirical study (see Section 4.5). This empirical study is divided in two parts: a static analysis that tracks comments in methods searching for possible references of collections usage; and a dynamic analysis that traces homogenous collections *i.e.* their elements are instances of the same class (different than Object) or its subclasses. In the static analysis we find that 3 out of 10 documented usages of collections clearly specify the kind of collection. Certainly, the use of parametric types would benefit those usages. In the dynamic analysis, we found that 97.9% of collection objects are homogenous. This confirms that generics would indeed be beneficial for Smalltalk programmers. Additionally, the introduction of generics will not affect programmers negatively, because they can still use raw types (*e.g.* Array<Dyn>) in their programs without the penalty of having to write explicit casts. Hence, we include generics in Gradualtalk.

3.4.3 Summary

Table 3.3 lists the Smalltalk features and idioms (first column) with their corresponding typing features (second column) that cover them. This table is divided in three groups: language features, programming idioms and typing idioms. Almost all language features and idioms are covered, with the exception of *reflection*, *type-based dispatch* and *symbol as method*. We arrive at these decisions after carefully discussing each language feature, idiom and typing extension. We performed two large-scale and three preliminary empirical studies, and reviewed existing literature about each feature and idiom to make informed decisions based on proper evidence.

Feature or Idiom	**Covered by**
Class-based objects	Nominal subtyping
Classes as objects	Self types (w/ extensions)
Lambdas	Gradual typing
Control structures	Gradual typing
Returning self	Self types
Traits	Structural subtyping
Live programming[+]	Incremental (type) checking
Open classes[+]	Incremental (type) checking
Reflection[□]	Effect system[*]
Abstract methods	Gradual typing (w/ small modification)
Inappropriate methods	Gradual typing (w/ small modification)
Joining values	Union types
Type-based dispatch[□]	Flow-sensitive typing[*]
Symbols as methods[□]	–
Explicit coercion	Casts
Collections	Generics

Table 3.3: A summary of covered language features and idioms by the base type system and typing features.

[+] The feature can also be covered by an effect system[*].
[□] The language feature or idiom will not be covered in the first version of Gradualtalk.
[*] We conclude that the typing feature will not be added to Gradualtalk.

Chapter 4

Preliminary Empirical Studies

This chapter presents a series of empirical studies specially designed to support the design decisions of Gradualtalk. We start by giving a general introduction, an overview of the studies and the contribution of this chapter (Section 4.1). We then describe the corpus and the methodology that all studies share (Section 4.2). We then present all three empirical studies: the use of `self` as a return value (Section 4.3); the use of joining values (Section 4.4); the use of collections (Section 4.5). For each study, we explain our motivation for the study, describe its particular methodology, and present the related results. In Section 4.6, we detail the threats to validity starting with the common threats and later, those specific to each study. Finally, we conclude recalling their impact on the design of Gradualtalk (Section 4.7).

4.1 Introduction

The introduction of types in dynamically typed languages benefits programmers in several aspects. For example annotated types increase both the overall documentation and the reliability of programs. However developing retrofitted type systems is problematic, because such type systems must fit the dynamically typed language, *i.e.* supporting language features and idioms as much as possible. Therefore, the design of a retrofitted type system requires careful analysis on the design decisions. We argue that those design decisions demand a proper justification with empirical evidence.

In this chapter, we present three empirical studies that help us make informed decisions on the design of Gradualtalk, a retrofitted type system for Smalltalk. We analyze (in each study) a corpus of 139 open source Smalltalk projects that totals 651,343 lines of code. The empirical studies are specially designed to track and measure the prevalence of some language features and idioms, such as:

Returning self In Smalltalk, methods return `self` (*i.e.* the current host object) by default. This allows programmers to write more fluent object interfaces by chaining method calls on the same object—*i.e.* method chaining. For the perspective of a type system, this language feature requires the introduction of self types to be properly handled. However, if programmers do not often return `self` then self types may not be a priority feature to implement in a retrofitted type system. Therefore, we perform an empirical study on the use of `self` as a return value. This a study helps us make informed decisions on whether self types is necessary or not.

Joining values dynamically typed languages, such as Smalltalk, allow programmers to use variables that have any of several representations. For example a variable that may be a character on an execution path and a number on another. Another example are the methods that return different kinds of values depending of their inputs. Such programming idiom requires union types to be properly typed in a retrofitted type system. However if this idiom is rarely used, we can safely ignore it. Therefore, we perform an empirical study on the use of joining values. This study helps us understand the prevalence of this idiom and the relevance of union types.

Collections The Collection API is one of the most used APIs in mainstream languages, including Smalltalk. Collection APIs are highly tested to guarantee their reliability. The introduction of types, especially generic types, can increase the reliability of collections. However generic types requires that programmers not only annotate collections declarations, but also collections usages. However, if annotating collection usages requires a significant effort, we can arguably conclude that generic types is not necessary at least for collections. Therefore we perform an empirical study on the use of collections.

This study helps us understand the potential and possible issues related to generic typed collections.

These studies help us make informed decisions on the design of Gradualtalk. Additionally, these results inform practitioners about the prevalence of such language features and idioms in practice. We argue that these studies and their conclusions make the design of Gradualtalk stronger. Hence we expect a type system that will be practical and usable for most Smalltalk programmers.

4.2 Experimental setup

In this section, we describe and analyze the corpus, as well as the methodology that we follow to inspect the data and answer the research questions.

Corpus The corpus for these studies is composed of open source Smalltalk projects from the standard Pharo development image (version 1.2.1), the web framework Seaside (version 3.0.4), and related sub-projects, *e.g.* JQuery for Seaside. In total we analyze 139 projects, 4,979 classes, 76,137 methods and 651,343 lines of code. We believe these projects are a good representation of system and application projects in Smalltalk. This is because the corpus includes projects that range from core Smalltalk packages, like Kernel, to industry targeted projects, like Seaside.

Methodology The methodology that we follow for each empirical study is the same and shares the above corpus. We start by defining the specific research question and its relevance for Gradualtalk and practitioners in general. Based on the research question, we define the relevant data for each study, and describe a pattern for tracing them or a metric for classifying them. We then apply a static and/or dynamic analysis based on the required data for the study. In the case of the static analysis, we inspect the source code (class and method definitions) looking for specific text and AST patterns. For the dynamic analysis, we inspect the live objects in a current running image and classify them according to the

study metrics. Finally, we discuss the results and conclude with the impact of these results on both the design of Gradualtalk and Smalltalk community.

We built a simple framework[1] to perform the static analyses in Smalltalk. Initially, we preloaded the Seaside packages in the Pharo image. We then reflectively collect all packages, classes and methods available in that image using the standard reflective APIs of Smalltalk. We then traverse all of them searching for text or AST patterns in the class and method definitions. Finally, we collect and classify all classes and methods that matched the patterns. In the dynamic analysis, we collect all instances of a given class and its subclasses using the reflective API. We use as a benchmark, a running image that has all standard Pharo services and Seaside web services running. However, we only applied the dynamic analysis in a single study (see Section 4.5).

4.3 On the use of `self` as a return value

Smalltalk is an object oriented language, whose core is small compared to other mainstream languages such as Java. Proof of this is that Smalltalk avoids unnecessary keywords. A classical example is void or unit; these keywords are used in other languages to represent the absence of a return value. Instead, in Smalltalk all methods return self (*i.e.* the current host object) by default. Hence, all methods always have a return value.

This particular feature allows programmers to chain method calls on the same object, therefore reducing unnecessary code, and making the object interface more fluent. For instance:

```
figure setDefaultColor.
figure computeShape.
figure draw: graphics.

"Using method chaining"
figure setDefaultColor computeShape draw: graphics.
```

In the above example, the three method calls on figure can be condensed in a single line using method chaining. Method chaining is also used with methods that do not return self. However those usages are not relevant for our analysis.

[1] http://ss3.gemstone.com/ss/SimpleInspector

Although returning self makes possible the benefits of method chaining, it also implies some challenges for type systems. This is because self is bounded dynamically—*i.e.* at runtime—hence a traditional type system cannot statically determine a concrete type for it. Nevertheless, the introduction of Self types [Saito and Igarashi, 2009] would allow type systems to properly type self in Smalltalk, and allow programmers to continue using method chaining. However, is self as a return value pervasively used in Smalltalk? In other words, if programmers do not often return self in their code, the introduction of Self types may not be necessary. Therefore, we formulate the following research question:

RQ_s: How prevalent is self as a return value? Answering this question will help us make informed decisions about Self types in Gradualtalk. Additionally, this can help us measure the potential (the upper bound of possible usages) of method chaining on the same object in Smalltalk.

In this section, we perform a preliminary empirical study of the use of self as return value in Smalltalk to answer the above research question. We statically analyze the corpus (Section 4.2) searching for explicit and implicit use of self as return value. These results help us understand the relevance of Self types for a retrofitted type system for Smalltalk, and the potential of method chaining.

4.3.1 Methodology

We statically trace the return values of all methods available in the corpus. Specifically, we search for AST return statements in the method body. We then classify the methods into three groups:

Implicit: Those that do not include the return statement. They return self implicitly. Classical examples are setter methods:

```
Person>> name: aString
  name := aString
```

Explicit: Those that return self explicitly (and literally) in any of their branches. Methods can have several return statements in their body. We consider that a

method returns self explicitly, if at least one of those return statements is self. For example, from the corpus:

```
RefactoryChangeManager>> undoOperation
    undo isEmpty
        ifTrue: [ ↑ self ].
    self ignoreChangesWhile: [
        | change |
        change := undo removeLast.
        redo add: change execute ]
```

In the above example, self is returned explicitly forcing the method exit, because the operation stack undo is empty.

Non-self: This is the group for all remaining methods. They represent methods that return any other value or complex expression that is not literally self. For instance:

```
Person>> name
  ↑ name
```

Getters are the classical example, however there are other examples, such as computing a value and returning it.

The above groups categorize all methods in the corpus. However, only for the implicit and explicit groups do a Self type approach makes sense.

4.3.2 Results and discussion

Figure 4.1 shows the distribution of methods based on their return statements. This shows that almost half of the methods return self (45.8%—34,904 out of 76,137). From them, an overwhelming percentage are implicit returns (92.3%—32,207), while a minority are explicit returns (7.7%—2,697). These results show the relevance of self as a return value. Additionally, this study suggests that Smalltalk programmers return self explicitly only in specific cases, and hence being self as the default return in Smalltalk has a huge impact on the use of the language. Method chaining on the same object has the potential to be used with (almost) half the methods, hence it may confirm that object interfaces in Smalltalk are fluent. In the case of Gradualtalk and based on these results, Self types is a priority typing feature that must be included in the type system.

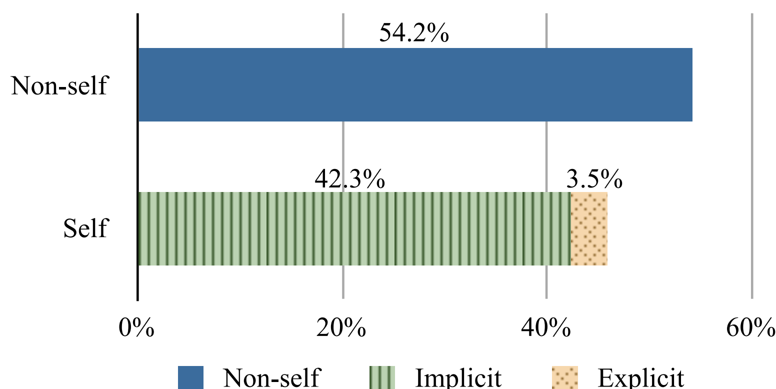

Figure 4.1: Method distribution based on their return.

4.4 On the use of joining values

dynamically typed languages are usually recognized as very flexible languages. This capability allows programmers to write prototypes and proof of concepts easily and quickly. Making use of such flexibility, we often find that a single variable may have any of several representations. For instance consider the following code:

```
"dict is a string to string dictionary"
myVar := dict lookUp: key.
```

where method lookUp: may return the string associated with key, if it exists in dict, otherwise it returns false. In this context, myVar may represent false (a boolean) or a string. The main goal of these values are to provide extra flexibility for programmers.

The use of joining values in statically typed languages requires the introduction of union types. A union type denotes a set union of some given types. In the particular case of Gradualtalk, the introduction of union types benefits programmers and allows the type system to properly type some methods in the Smalltalk Kernel API, such as control flow statements:

```
Boolean>> (a | b) ifTrue: (→ a) trueBlock ifFalse: (→ b) falseBlock
 ...
```

where branches may return different types, *e.g.* a or b. However, if the prevalence of this idiom is only in specific scenarios or in the kernel API, we can arguably

implement an ad-hoc version of union types for those cases. Therefore, we formulate the following research question:

RQ_u: How frequently are programmers joining several values in a single variable or method name? Answering this question will help us understand the relevance of union types for Gradualtalk. Additionally, this will inform practitioners about the presence of this idiom in practice.

In this section, we perform a preliminary empirical study on the use of variables and method names (*aka.* selectors) that may have any of several representations. We statically analyze the corpus in Section 4.2 seeking for the use of the above idiom in variables and selectors. These results help us make informed decisions regarding union types for Gradualtalk.

4.4.1 Methodology

We statically trace potential usages of this idiom. Specifically, we look for method names at method declarations, and variable declarations (temporary variables, and method and block parameters) that match the pattern *xOrX*, where x and X are a lower and upper case character respectively. Some examples (from the corpus) are the variable aBlockOrString and the selector asBinaryOrTextStream. We then manually check all variables and selectors to discard false positives, *i.e.* those where the left or right hand side of the Or cannot be related to a type, *e.g.* a Smalltalk class. Some examples of false positives are variable leftOrRight and selector againOrSame:. We have the confidence that our heuristic may refer to usages of the tracked idiom.

4.4.2 Results and discussion

The static analyzer was able to detect 1,380 declarations of variables and methods that match the xOrX pattern. After a manual inspection, we discarded 169 (12.2%) of those declarations, because their left or right hand side of the Or cannot be related to types. This low number of false positive confirms that our heuristic is good enough to detect usages of the tracked idiom. For the remainder of the study, we use the refined declarations that account for a total of 1,211 declarations.

Additionally, we found that the most frequent variable name is aStringOrText with 97 occurrences. Note that String and Text classes are not related hierarchically, *i.e.* their least common ancestor is Object. Clearly, union types would have a significant impact on those usages.

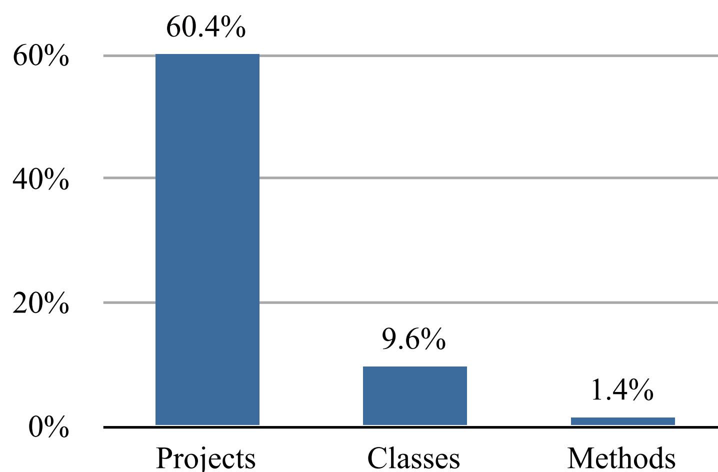

Figure 4.2: Presence of joining values in projects, classes and methods.

Figure 4.2 shows the presence of this idiom at the level of projects, classes and methods. At the project level, we find that 60.4% (84 out of 139) of projects use at least one variable or method declaration that matches the pattern. Although this percentage is high at the project level, this is not the case at other levels. At the level of class, we find that almost one out of ten classes 9.6% (477) use the tracked idiom. Finally, at the method level, we find that a total of 1,035 methods (1.4%) use such variables or selectors, with an average of 1.17 of these declarations per method. These results suggest that this idiom is somewhat rarely used, however it is still significant enough to argue that the idiom exists in practice, and hence retrofitted type systems may take it into account. Furthermore, union types implementation does not require a significant effort, *e.g.* it requires some straightforward subtyping rules and computing structural intersection between types. Therefore, union types would be beneficial for Gradualtalk with a low implementation impact.

4.5 On the use of collections

Collection is a general term that refers to common data structures. Some well-known and classical data structures are arrays, lists, sets and hashmaps. In the particular case of Smalltalk, the Collection API also includes strings and streams. The Collection API is one of the most used APIs in mainstream languages. All (or almost all) programs require some data structure to work property. Therefore, they must be completely reliable, *e.g.* free of bugs. This is why Collection APIs are usually highly tested. The introduction of types, especially generic types, can increase the reliability of collections.

Collections usually store elements of the same type, *e.g.* instances of the same class. However, in dynamically typed languages, programmers can abuse this and store any kind of elements. This complicates the design and use of retrofitted type systems. Type system designers have to add extra flexibility on generic types. Programmers are forced to use raw collections (*e.g.* Array instead of Array<String>) or complex parametric types (*e.g.* Array<String|Text|Stream>). In that sense, if too many collections are heterogeneous, the introduction of generics may not benefit programmers. Therefore understanding the use of collection from the point of view of generic types is important for the design of a practical retrofitted type system. Consequently, we formulate the following two research questions:

RQ_c1: How often are programmers referring to specific kinds of collections in method comments? Each one of those comments may indicate a programmer's intention of using generic typed collections. This can be seen as a lower bound estimation of potential generic typed collections. Consequently, answering this question will help us understand the potential of generic types in collections.

RQ_c2: Are collections homogenous or heterogeneous? Answering this question would shed light on the use of heterogeneous collections in practice. This can be seen as an upper bound estimation of possible complex usages of generic types that use collections. This will inform both type system designers about the pragmatic introduction of generics in Gradualtalk, and practitioners on the

prevalence of heterogeneous collections.

In this section, we perform a preliminary empirical study on the use of collections. First, we statically analyze the corpus (Section 4.2) searching for commented methods that may refer to concrete or raw use of collections. Second, we dynamically analyze the collection objects in the running image of the corpus to determine if they are homogenous or heterogeneous collections. These results will help us make informed decisions regarding generics in Gradualtalk.

4.5.1 Methodology

The empirical study is divided in two parts: a static and dynamic analysis. In the static analysis, we track regular expressions in source code comments. On the other hand the dynamic analyzer collects live collection objects in a running image. Each of these analyses has its own methodology detailed in the following paragraphs:

Static analysis. We build a static analyzer that tracks comments in methods for possible references of collections uses. In particular, we search for words like array, collection, sequence, map, list and dictionary. We use regular expressions to detect the above keywords in the right context. We classify those usages into two main groups:

Raw: A raw collection reference is a comment that refers to a collection, but the elements inside it are not clearly identified. For instance, from the corpus, *"a collection with equal elements"*, it is not clear what kind of elements are inside the *"collection"*.

Concrete: A concrete collection reference is a comment that refers to a collection and makes it clear what kind of collection it is. One example from the corpus is *"Return the list of all current ChangeSets"*. We argue that comments in this group indicate a possible use of parametrized collection, *i.e.* a parametrized generic type, with a clear reference of what that parametrized type would be.

For our analysis, we take a conservative approach assuming that just concrete collection comments make a clear reference to generics.

Dynamic analysis. In the dynamic analysis, we collect all collection objects in the running image of the corpus, where all standard Pharo services and Seaside web services are running. We use the standard reflective APIs for gathering all instances of Collection classes, *i.e.* Collection and subclasses. We then classify those objects into three groups:

Strict-homogeneous: Strict-homogenous collections are collections where all of its elements are instances of the exact same class. This definition excludes elements that are instances of subclasses. Hence the collection { #a . #b . 'c' } is not strict-homogeneous, while the collection { #a . #b . #c } is. This is because 'c' is an instance of String, while #a and #b are instances of Symbol.

Relaxed-homogeneous: A relaxed-homogenous collection is a collection where all of its elements are instances of the same class (different than Object) or any of its subclasses. An example is { #a . #b . 'c' } because all elements are instances of class String and its subclass Symbol. In other words, relaxed-homogenous collections have elements of classes whose least common ancestor is different than Object.

Heterogeneous: This group is the complementary group. It groups all collections that contain elements of different classes whose least common ancestor is Object. Hence, { #a . #b . 1 } is an heterogeneous collection, as well as { 1 . 2 . true }, because both have Object as their least common ancestor.

As different results can be gathered at different times of inspection; we repeated the data collection 30 times at different stages of the execution, *e.g.* http sever off, running several Seaside applications, running unit tests, etc. We then compute the average for the final result. For generics, heterogeneous collections may complicate its use. However some typing techniques, such structural and union types, can help programmers flexibly annotate usages of generic types. In that sense, this analysis over-estimates the issue of complex generic types when using collections.

4.5.2 Results and discussion

In this section, we present the results of the static and dynamic analysis. We then discuss their implications in the design of Gradualtalk.

Collections references in method comments

The static analyzer detects a total of 4,134 commented methods that refer to collections. Figure 4.3 shows the distribution of those methods. Raw collection references account for a total of 2,912—70.2%. On the other hand, concrete collection references are 1,231 or 29.8%. This result suggests that 3 out of 10 documented usages of collections clearly specify the kind of collection. Certainly, the use of parametric types would benefit at least those usages.

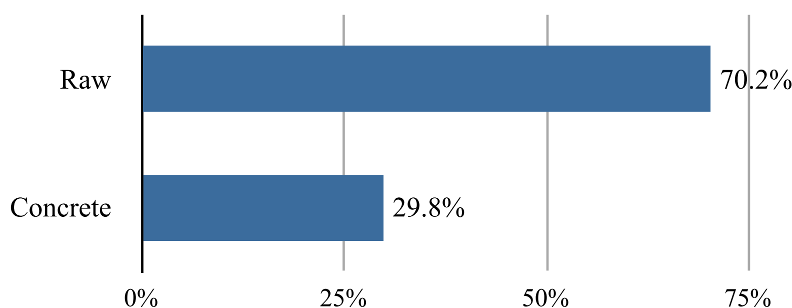

Figure 4.3: Distribution of commented methods about collections.

Homogenous versus heterogeneous collections.

In the dynamic analysis, we inspect live collection objects, *i.e.* objects that are instances of Collection or any of its subclasses, at 30 different times in the running image. We collect in total 6,536,818 collection objects with an average of 217,893.9 collections that store more than 15 million of elements per inspection. Figure 4.4 shows the distribution of those objects. We find that an overwhelming 87.2% (5,701,922—190,064.1 in average) are strict-homogenous collections and 10.7% (700,429—23,347.6 in average) are relaxed-homogenous collections. On the other hand, just 2.1% (134,468—4,482 in average) are heterogeneous collections.

This result makes sense, because handling heterogeneous collections usually requires more code, *e.g.* adding checks for disambiguating the type of each element. Finally, this result suggests that generics may have a good impact on Collections API with low complexity for programmers.

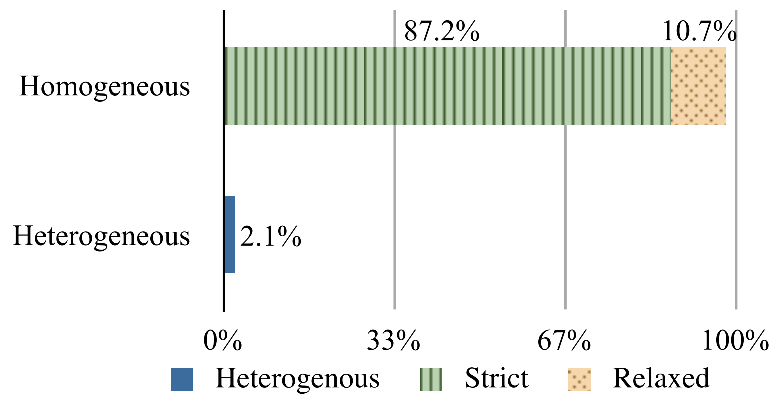

Figure 4.4: Distribution of collection objects in average per inspection.

Discussion.

We perform two analyses to understand the relevance and the pragmatic introduction of generics in Gradualtalk. In the static analysis, we find that 29.8% of method comments that refer to collections are concrete collection references. The introduction of generic types allows programmers to directly annotate the kind of collection used instead of write it in the comments. Hence those concrete collection references are clear candidates for the use of generics. This suggests a lower bound of potential generic typed collections. In the dynamic analysis, we find that 97.9% of collection objects in a running image are homogeneous (87.2% strict and 10.7% relaxed). This overwhelming percentage confirms that the use of generics will not require complex types to be practical in Smalltalk. This suggests an upper bound of possible complex generic types that use collections. These results suggest that the introduction of generic types benefit the use of the Collection API in Smalltalk. Therefore, based on these results, generic types are a relevant and pragmatic feature for Gradualtalk.

4.6 Threats to validity

In this section, we detail the threats to validity for all empirical studies performed in this chapter. Some threats are common to all studies, while others are specific to each study.

Common threats.

All empirical studies share the same corpus. This corpus is a collection of open source Smalltalk projects. Although the corpus includes some industry focus projects such as Seaside, we cannot generalize our results to close-source industrial projects. Additionally, the source code come from the Pharo and Squeak community. Therefore, we do not know whether the results would be similar or not in other communities, such as VisualWorks.

On the use of `self`.

Some methods classified in the group non-self may actually return `self` (*e.g.* in an aliased variable). In this scenario the results are just an underestimation of the actual use of `self`. On the other hand, some methods in the explicit group may never return `self` at runtime, because the returning-self branch is never executed. However detecting those cases requires a dynamic analysis. Nevertheless most methods returning self are in the implicit group (92.3%) rather than in the explicit group (7.7%).

On the use of joining values.

Although we did our best to detect usages of this idiom through the pattern `xOrX`, detected variables and selectors may not represent the union of several types. However, we additionally perform a manual check to discard those cases. The low number of false positives (169—12.2%) confirms that our heuristic is good enough to detect usages of the tracked idiom. Additionally, other source code entities, such as variable names (*e.g.* `strings`), can be used to more precisely compute the relevance of this idiom. However, we focused on method comments, because we take the assumption that programmers who writes comments are more willing to annotate types than those that do not comments their methods.

On the use of collections.

Method comments that mention collections may not use any collection in the method body. However, this would imply that comments are wrongly written, which is very unlikely. In a manual inspection of 10 methods (randomly selected), we found, that in all, their comments are related to their bodies. Additionally other source code entities, such as variable names, may be used to more precisely compute the relevance of the tracked idiom. However, we focused on method comments, because we take the assumption that programmers who write comments are more willing to annotate types than those that do not comment on their methods. Another threat is that we trace only comments in English. This is because the tracked keywords (*e.g.* array) are in English. Though our corpus is mostly in English, we expect that comments in other natural languages are a minority.

Dynamic analyses are very dependent on the executed benchmarks. In our case, the dynamic analysis depends on the moment where we inspect the collection objects. This is because at different times, there could be more or less collection objects. This can affect the overall results of homogenous and heterogeneous collections. In order to mitigate this threat, we inspect collection objects at 30 different stages of the Pharo image execution. Some of these stages are http server off, running multiple unit test suites, running several Seaside applications, etc.

4.7 Conclusions

We performed three empirical studies on the use of `self` as a return value (Section 4.3), the use of variables and selectors that have any of several representations (Section 4.4), and the use of collections (Section 4.5). We statically and dynamically analyse a corpus of 139 Smalltalk projects that range from core system packages such as Kernel to industry focus projects such as Seaside.

One of the main contributions of this chapter is that we answered four research questions that help us make informed decisions on the design of Gradualtalk. In particular, these studies help us understand the typing features:

Self types Almost half the methods return self, which clearly highlights the need for self types in Gradualtalk. We also find that most of those methods return self implicitly. This tells us that the Smalltalk's design choice of returning self by default makes object interfaces more fluent. These results confirm that self types are a must-have feature for Gradualtalk.

Union types We find 1,211 declarations of variables and selectors that may require the use of union types. Those declarations are present in 1.4% of all methods in the corpus. Although this result suggests an occasional use of union types, we cannot avoid them. Furthermore, considering a low impact on the implementation of union types in general, we consider union types as a beneficial extension for Gradualtalk

Generic types We conducted two analyses to understand the potential of generic types and their possible issues in practice. In the first analysis, we find that 3 out of 10 method comments about collections are clearly annotating the kind of collection used. This may indicate a lower bound of the potential of generic types when using collections. In the second study, we find that only 2.1% of collection objects are heterogeneous. This may define an upper bound of possible complex usages of generic types that use collections. These results suggest that generic types is a relevant and pragmatic extension for Gradualtalk.

We believe that these results and discussion make the design of Gradualtalk stronger and practical.

Chapter 5

How and Why Developers Use the Dynamic Features of Smalltalk

The dynamic and reflective features of programming languages are powerful constructs that programmers often mention as extremely useful. However, the ability to modify a program at runtime can be both a boon—in terms of flexibility—, and a curse—in terms of tool support. For instance, usage of these features hampers the design of type systems, the accuracy of static analysis techniques, or the introduction of optimizations by compilers.

In this chapter, we perform an empirical study on a large Smalltalk codebase, in order to assess how many dynamic and reflective features are actually used in practice, whether some are used more than others, and in which kinds of projects. In addition, we performed a qualitative analysis of a representative sample of usages of dynamic features in order to uncover (1) the principal reasons that drive people to use dynamic features, and (2) whether and how these dynamic feature usages can be removed or converted to safer usages. These results are useful for making informed decisions about which dynamic and reflective features to consider when designing Gradualtalk, see Section 3.4.2.

This chapter is structured as follows: First we give an overall introduction and present our four hypotheses for the empirical study (Section 5.1). We then describe our experimental methodology, analysis infrastructure, and the dynamic

features we look in Section 5.2. We then present our first results per hypothesis in Section 5.3. We then discuss these results and their implications per hypothesis (Section 5.5). Later, we perform our qualitative analysis of dynamic feature usages and identify refactorable occurrences (Section 5.6). We then discuss the potential threats to the validity of this study (Section 5.7), before reviewing related work (Section 5.8). Finally, Section 5.9 concludes with recommendations for designing Gradualtalk with regard to effect types and incremental type checking features.

5.1 Introduction

Dynamic object-oriented languages such as Smalltalk [Goldberg and Robson, 1983] or Ruby allow developers to dynamically change the program at runtime, for instance by adding or altering methods; languages such as Java, C# or C++ provide reflective interfaces to provide at least part of the dynamism offered by dynamic languages. These features are extremely powerful: the Smalltalk language for instance ships with an integrated development environment (IDE) that uses these dynamic features to create, remove, and alter methods and classes while the system is running.

If powerful, these dynamic features may also cause harm: they make it impossible to fully check the types of a program statically; a type system has to fall back to dynamic checking if a program exploits dynamic language features. Until recently, the problem of static analysis in the presence of reflection was largely sidestepped; current solutions to it fall back on dynamic analysis in order to know how the dynamic features are exercised at runtime [Bodden et al., 2011]. Another example is the (static) optimization of program code, which is impossible to achieve for code using any dynamic language feature. Moreover, tools are affected by the use of these features. For instance, a refactoring tool may fail to rename all occurences of a method if it is used reflectively, leaving the program in an inconsistent state. In short, dynamic language features are a burden for language designers and tool implementors alike.

This problem is exacerbated since language designers and tool implementors do not know how programmers are using dynamic language features in practice. Dynamic features might only be used in specific applications domains, for instance

in parsers/compilers, in testing code, in GUI code, or in systems providing an environment to alter code (eg. an IDE). Having precise knowledge about how programmers use dynamic features in practice, for instance how often, in which parts, and in which types of systems they are used, can help language designers and tool implementors find the right choices on how to implement a specific language extension, static analysis, compiler optimization, refactoring tool, etc. If it turns out that a given dynamic feature is used in a minority of cases, then it may be reasonable to provide a less-than optimal solution for it (such as resorting to dynamic type checking in a static type system). On the other hand, if the usage is pervasive, then a much more convincing solution needs to be devised. Hence, it is of a vital importance to check the assumptions language designers and tool implementors might have against reality.

In this chapter, we perform an empirical study of the usage of dynamic language features by programmers in Smalltalk. We survey 1,000 Smalltalk projects, featuring more than 4 million lines of code. The projects are extracted from Squeaksource (http://www.squeaksource.com), a software super-repository hosting the majority of Smalltalk code produced by the Squeak and Pharo open-source communities [Lungu et al., 2010]. We statically analyze these systems to reveal which dynamic features they use, how often and in which parts. Next, we interpret these results to formulate guidelines on how language designers can best deal with particular language features, depending on how (frequent) such features are used in practice.

In addition to these quantitative results, we also performed a qualitative analysis of a representative sample of 377 usages of dynamic features across our corpus. By focusing on this restricted data set, we were able to perform a deeper analysis, with two goals. The first goal is to investigate the principal reasons why developers use dynamic features, in order to pinpoint areas of applications, and types of computations that are more prone to these usages than other. Understanding these practices can also indicate ways to extend a programming language to support certain patterns in a more robust manner. The second goal is to determine whether some of these dynamic usages can be removed (or converted to safer usages of the same feature), in order to reduce the extent of the problem and simplify static analyses. Thus, we provide guidelines on how to refactor some

of the common idioms we encountered.

Contributions. This chapter explores the usage of dynamic features in Smalltalk in order to gain insight on the usage of these features in practice.

Our first contribution is a quantitative analysis of a large corpus of source code (1,000 Smalltalk projects) in order to validate, or invalidate, the following hypotheses:

A1 Dynamic features are not used often. More precisely, we are interested in determining which features are used more than others.

A2 Most dynamic features are used in very specific kinds of projects. We conjecture that they are more often used in core system libraries, development tools, and tests, rather than in regular applications.

A3 The specific features that have been integrated in more static languages over time (eg. Java) are indeed the most used.

A4 Some usages of dynamic features are statically tractable, unproblematic for static analyses and other tools.

While our study allows us to validate these hypotheses, we do so with some caveats; this makes it still necessary for language designers, tool implementors, and developers of static analyses to carefully consider dynamic features. We provide prelimary guidelines as to which are most important.

Since dynamic features cannot be ignored outright, our second contribution is the qualitative analysis of a representative sample of 377 dynamic feature usages in order to better understand the reasons why developers resort to using these dynamic features, and whether some of them can be refactored away, or converted to safer usages of the same features. We find that some of the usages are unavoidable, others are due to limitations of the programming language used, some can be refactored, and others are mostly superfluous.

As a consequence of these two studies, we gain insight into how language designers and tool providers have to deal with these dynamic features, and into why the developers need to use them—yielding further insights in the limitations of programming languages that need to be addressed.

5.2 Experimental setup

To find out how developers use the dynamic features provided by Smalltalk in practice, we perform an analysis of a large repository of Smalltalk projects. This section describes the experimental setup, that is, the methodology applied to perform the analysis, the analysis infrastructure, and an explanation of the dynamic features we are analyzing. This and the following sections focus only on the quantitative analysis; the qualitative analysis of a representative sample of dynamic feature usages is described entirely in Section 5.6.

5.2.1 Methodology

We started our analysis by looking at all 1,850 software projects stored in Squeaksource in a snapshot taken in early 2010. We ordered all projects by size and selected the top 1,000 projects, in order to exclude small or toy projects. Squeaksource was the *de facto* source code repository for open-source development in the Squeak and Pharo dialects at the time we analyzed the projects[1]. We believe this set of projects is representative of medium to large sized Smalltalk projects originating from both open-source and academia. Table 5.1 summarizes the top ten projects sorted by lines of code (LOC), and also shows number of classes and methods. The last row shows the total for the 1,000 projects analyzed in this study.

In order to analyze the 1,000 projects, we developed a framework[2] in Pharo[3] to trace statically the use of dynamic features in a software ecosystem. This framework is an extension of *Ecco* [Lungu et al., 2010], a software ecosystem model to trace dependencies between software projects. Our analyzer follows three principal steps: *Trace*, *Collect* and *Classify*.

To *Trace*, first the analyzer reads Smalltalk package files from disk and builds a structure (an ecosystem) which represents all packages available on disk. Later, the analyzer flows across the ecosystem structure parsing all classes and methods from

[1] Currently, other repositories like Smalltalkhub (http://www.smalltalkhub.com) and Squeaksource3 (http://ss3.gemstone.com) are mainly used by the community; most of the projects in these repositories are simply updated versions of the ones that are in our corpus.

[2] Available at http://www.squeaksource.com/ff

[3] http://www.pharo-project.org

Project	**LOC**	**Classes**	**Methods**
Morphic	124,729	676	18,154
MinimalMorphic	101,190	483	13,887
System	91,706	502	10,970
Formation	89,172	695	9,833
MorphicExt	69,892	236	9,461
Balloon3D	68,020	397	7,784
Network	58,040	447	8,207
Collections	55,254	405	9,093
Graphics	52,837	139	5,267
SeaBreeze	47,324	228	3,466
Total (1,000)	**4,445,415**	**47,720**	**652,990**

Table 5.1: The 10 largest projects in our study.

each package. In the method parsing process, the analyzer traces statically all calls of the methods that reflect the usage of dynamic features in Smalltalk. Section 5.2.3 describes these dynamic features in more details and lists the corresponding method names.

The *Collect* step gathers the sender, receiver and arguments of each traced message call AST nodes. The collected data is stored in a graph structure, which recursively catalogs the sender into packages and classes, and the receiver and arguments into several categories: literals (*e.g.* strings, nil, etc), local variables, special variables (*i.e.* self or super), literal class names, and arbitrary Smalltalk expressions.

The third step, *Classify*, is performed in the graph structure. Each call site is classified either as *safe* or *unsafe*: A safe call site is one for which the behavior can be statically determined (*e.g.* the receiver and arguments are literals, for instance), whereas an unsafe call may not be fully statically determined. The exact definition of what is safe and unsafe depends on each feature, as described in Section 5.2.3.

Characterizing usages as safe or unsafe is an indicator of *how* dynamic features are used, and how challenging it may be for a static analysis or development tool to support it. This study also answers the *where* question: which kinds of projects make use of these features. For this, we introduce project categories, described below.

5.2.2 Project categories

In order to characterize in which kinds of projects dynamic features are used, we classified each project according to five categories:

- System core (**System, 25 projects**): Projects that implement the Smalltalk system itself.
- Language extension (**Lang-Ext., 55 projects**): Projects that extend the language, but are not part of the core (eg. extension for mixins, traits, etc.).
- Tools and IDE (**Tools, 63 projects**): Projects building the Smalltalk IDE and other IDE-related tools.
- Test suites (**Tests, 24 projects**): Projects or parts of projects representing unit and functionality tests[1].
- Applications (**Apps, 833 projects**): Conventional applications, which do not fit in any of the other categories; this is the overwhelming majority.

5.2.3 Analyzed dynamic features

We consider four groups of dynamic features of Smalltalk in this study: first-class classes, behavioral reflection, structural reflection, and system dictionary. We came to this classification by iterating over a sample of the usages of the features in order to delineate their intent. This process was supported by our own experience in using the features. Non-standard and seldom-used features were omitted. In each of these groups, the use of the different features is identified by specific selectors, which we have identified based on our experience as Smalltalk developers. In addition to describing each feature and its corresponding selectors, this section explains how specific usages are characterized as safe or unsafe.

[1] All subclasses of TestCase are considered to represent tests, no matter how the rest of the project is categorized.

First-class Classes. This category includes features that are related to the usage of classes as first-class objects. As opposed to other object-oriented languages such as Java, Smalltalk classes can be receivers or arguments to methods. The use of first-class classes complicates matters for static analysis especially with respect to instance creation and class definition, as one cannot know which class will be instantiated or created.

Instance Creation. In Smalltalk, the typical instance creation protocol consists of new, which may be overridden in classes, while basicNew is the low-level method responsible for actually creating new instances. When tracing all occurrences of invocations of basicNew in the analyzed projects, we consider only two kinds of occurrences to be unsafe:

```
x basicNew.
(z foo) basicNew.
```

In the first case the receiver is a local variable, and in the second case the receiver is the result of a method invocation, or more generally, any arbitrary expression. Usages of basicNew with a literal class name or the pseudo-variables self or super as receiver are considered safe. Note that the type of self is statically tractable using self types, as in Strongtalk [Bracha and Griswold, 1993].

Class Creation. To create a new class, Smalltalk offers a range of subclass: methods that only differ in the arguments they accept. As for instance creation, we only consider a message send of subclass: to be unsafe if: (1) the receiver is a local variable or a complex Smalltalk expression, or (2) the argument (the class name to be created) is not a symbol. Examples of safe calls are:

```
Point subclass: #ColorPoint.
self subclass: #ColorPoint.
```

Examples of unsafe method calls are:

```
c subclass: #MySubClass.
Point subclass: x name.
```

The first example subclasses an undetermined class c, while the second example creates a subclass of Point with an undetermined name (the result of sending name to x).

Behavioral Reflection. Behavioral reflective features of Smalltalk allow programmers to change or update objects at runtime, or to dynamically compute the name of methods to be executed. We distinguish between the following features: object reference update, object field update, and message sending.

Object Reference Update. Selectors such as become: allow Smalltalk programmers to swap object references between the receiver and the argument. After a call to become:, all pointers to the receiver now point to the argument, and vice versa; this affects the entire memory. Determining at compile time, if this reference swap is safe or unsafe is challenging. We consider all calls to these selectors to be unsafe.

Object Field Read. In Smalltalk, object fields are private; they are not visible from the outside and must be accessed by getter and setter methods. The Smalltalk reflection API provides methods to access them by using their names or indexes, *e.g.* instVarNamed:. We categorize as safe usages of an "object field read" those with either a number, symbol or string literal as argument.

Object Field Update. Complementary to previous features, Smalltalk allows developers to reflectively change an object field value. For that propose, the Smalltalk reflection API offers methods such as instVarAt:put: and variants, to write into object fields without using the corresponding setter methods. We consider safe calls to be those where the object field index (the selector's first argument) is a number, symbol or string literal.

Message Sending. The perform: selector invokes a method by passing its name (a symbol) as the argument of the call, as well as the receiver object. This feature is also provided by the Java reflection API. Safe calls are those where the method name (the argument in the expression) can be determined statically—*i.e.* a symbol. In unsafe calls, the argument is a local variable or a composition of message calls (*e.g.* a string concatenation). Examples of unsafe calls are:

```
x perform: aSelector.
x perform: ('selectorPrefix', stringSuffix) asSymbol.
```

Structural Reflection. With the structural reflective features of Smalltalk, developers can modify the structure of a program at runtime by dynamically adding or removing new classes or methods. We consider the following structural reflective features:

Class Removal. In Smalltalk, classes can be removed from the system at runtime. We include this feature to analyze through the removeFromSystem selector where the receiver is the class to remove. In our analysis, we consider unsafe occurrences to be calls in which the receiver is a local variable, or a Smalltalk expression. Examples are:

```
c removeFromSystem.
(x class) removeFromSystem.
```

Superclass Update. Smalltalk programmers can change at runtime the behavior of a class by updating the superclass binding. This powerful feature is handled by superclass: selectors. Safe calls to them are those where both the receiver (the subclass) and the argument (the new superclass) are either a literal class name (including nil[1]) or self. Any other case is potentially unsafe. Safe examples are:

```
Point3D superclass: MyPoint.
self superclass: nil.
```

Method Compilation. Adding behavior at runtime allows programmers to dynamically load runnable code. Smalltalk provides selectors such as compile: to compile and add methods to a particular class. Calls where the argument—the code to be compiled, or the selector name—is lexically a string are safe; others are not. We further categorize safe calls to the compile selector in the following categories: *trivial*, simple code such as returning a constant or a simple expresion ; *getter/setter*, which returns/sets an instance variable; and *arbitrary* code—everything else.

Method Removal. This feature complements the one above, adding the ability to remove behavior from a class at runtime, with selectors such as removeSelector:. When tracing all occurrences of invocations of this kind of selectors, we categorize

[1] In Smalltalk the root superclass is nil.

those occurrences where the argument (the selector name) is a variable name or a composition of message calls (*e.g.* a string composition) as unsafe. Therefore, safe occurrences are when the argument is lexically a symbol. Example of unsafe occurrences are the following:

```
c removeSelector: aSelector.
c removeSelector: ('prefix' , varSuffix ) asSymbol.
```

System Dictionary. The Smalltalk system dictionary is a global variable (called Smalltalk), which registers all class definitions in the image. Smalltalk provides several methods to access, add, remove or alter class definitions. We focus only on usages of this dictionary that concern classes; other manipulations of the system dictionary concern global variables in general, and are hence out of the scope of this study. We distinguish between usages of this dictionary for reading, and writing. We also detect when aliases to this dictionary are created (*e.g.* by passing it as argument to a method).

Reading. The system dictionary provides several methods to both access class definitions and test for class existence. For example, at: returns the class object whose name is specified with the argument; hasClassNamed: verifies if the system defines a class with the given name. Safe usages are those where the class name (the argument) is a literal symbol or string. Unsafe expressions are those where the argument cannot be determined statically, for instance:

```
Smalltalk at: myVar .
Smalltalk hasClassNamed: ('Preffix' , suffixVar) .
```

Writing. The system dictionary can also be used to alter the classes defined in the system. For instance, some usages are:

```
Smalltalk at: #MyClass put: classRef .
Smalltalk removeClassNamed: #MyClass .
```

Both expressions are alternative ways of doing class renaming or removal operations, which are already feasible using the reflective abilities of Smalltalk presented previously. In our study, we classify as safe the usages where the argument in the call is a literal symbol or string, as shown in the previous example.

Aliasing. Because the system dictionary is a variable in Smalltalk, programmers can pass it around, and therefore aliases to this variable can be created. Aliasing makes it particularly difficult to track usages of the system dictionary. In this category, we collect direct aliasing (assignment expressions involving the system dictionary) as well as expressions that can yield aliases, *e.g.* passing the dictionary as an argument to a method or returning it in a block. All aliasing occurrences are categorized as unsafe usages.

5.3 Quantitative Results

This section presents the first results of the study. Initially, we present how many and how often projects use dynamic features. These results help us validate or invalidate hypotheses: A1, A2 and A3. We then analyze the usage of each dynamic feature in detail, which addresses hypothesis A4).

5.3.1 How programmers use dynamic features

A1: Are dynamic features used often? (see page 67)

In our analysis of the 1,000 projects, we found 20,387 dynamic feature occurrences, *i.e.* calls to a method implementing a dynamic feature. Only 11,520 methods use at least one dynamic feature; this shows that a fair proportion of methods either use a feature more than once or use several features at once. The 11,520 methods using dynamic features represent 1.76% of the 652,990 methods we analyzed for this study. This shows that use of dynamic features is punctual: most methods do not make use of them.

Of the total methods using dynamic features, 6,524 were in projects classified as "Applications" (1% of all analyzed methods) and 3,832 of these use dynamic features that we consider as unsafe (58.74% of the methods using dynamic features, or 0.59% of all methods). These results confirm the rarely use of unsafe dynamic features.

Projects classified as applications represent 83% of the projects, yet contain only 56.63% of the methods using dynamic features, confirming the fact that other project categories use these features more extensively. Of all the dynamic feature

usages, 12,094 were classified as unsafe (59.32%); 5,253 of those were in applications (43.43%).

A2: Do regular applications use fewer dynamic features? (see page 67)

To confirm assumption 2 (A2), which states that most dynamic feature usages occur outside application projects, we ran a statistical test comparing all application projects to all other projects. As the data of dynamic feature usages is normally distributed across projects, we use a Student's t-test (two-tailed, independent, unequal sample sizes, unequal variance) which yields a p-value of 0.00958 (d.f = 610.9, t = 2.599). This result confirms assumption 2.

However, the data stemming from non-application projects has a high variance and its mean number of dynamic feature usages differs considerably from those in application projects. Due to this, the effect size of this experiment is rather low (as expressed with a Cohen's d of 0.169), which would require us to analyze even more projects in order to reliably confirm assumption 2. Still, the current results are a strong indication that dynamic feature usage is more widespread in special projects such as language extensions or system core projects.

A3: What are the most prevalent dynamic features? (see page 67)

Figure 5.1 shows the distribution of the usage of dynamic features, with a maximum of 6,048 occurrences of message sending (29.67%) and a minimum of 114 occurrences for superclass updates (0.56%). Categories are distributed as follows: first-class classes with 15.46%, behavioral reflection with 44.41%, structural reflection with 15.85% and system dictionary with 24.28%. Four dynamic features—Message sending, Instance creation, Method recompilation, and Reading system dictionary—, account for more than 75% of the usages. Of these, Java provides three in its reflection API (message sending, instance creation, access to the class table), catering to 64% of the usages in the analyzed Smalltalk projects.

Figure 5.2 exhibits the per-feature distribution of all software projects arranged left to right in the following categories: *No Use*, projects with no occurrences of the analyzed feature (blue); *Safe*, projects that have one or more occurrences of the analyzed dynamic feature, but all occurrences are safe (green); *Unsafe in Systems,*

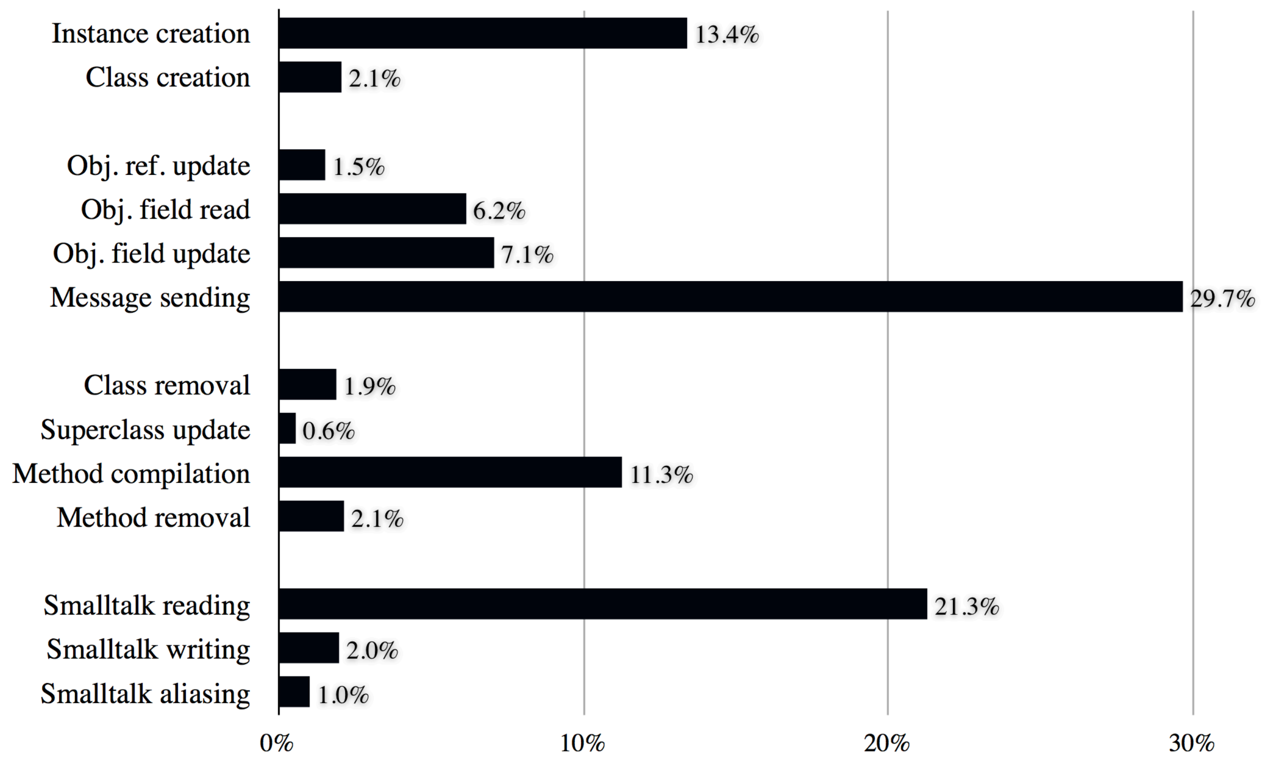

Figure 5.1: Distribution of dynamic feature usages.

Tests, Language extensions or Tools represents all projects in those project categories with at least one unsafe call of the feature (yellow); *Unsafe in Applications* includes application projects with at least one unsafe call (red). Most features (except instance creation, message sending and reading system dictionary) follow a common pattern:

- Many projects do not use the analyzed feature. This category ranges between 725 projects in method definition and 961 in the superclass update feature.

- Unsafe uses are almost equally distributed between applications and other categories, with an average of 45 and 53 projects respectively. Applications with 83% of the projects have comparatively fewer unsafe uses.

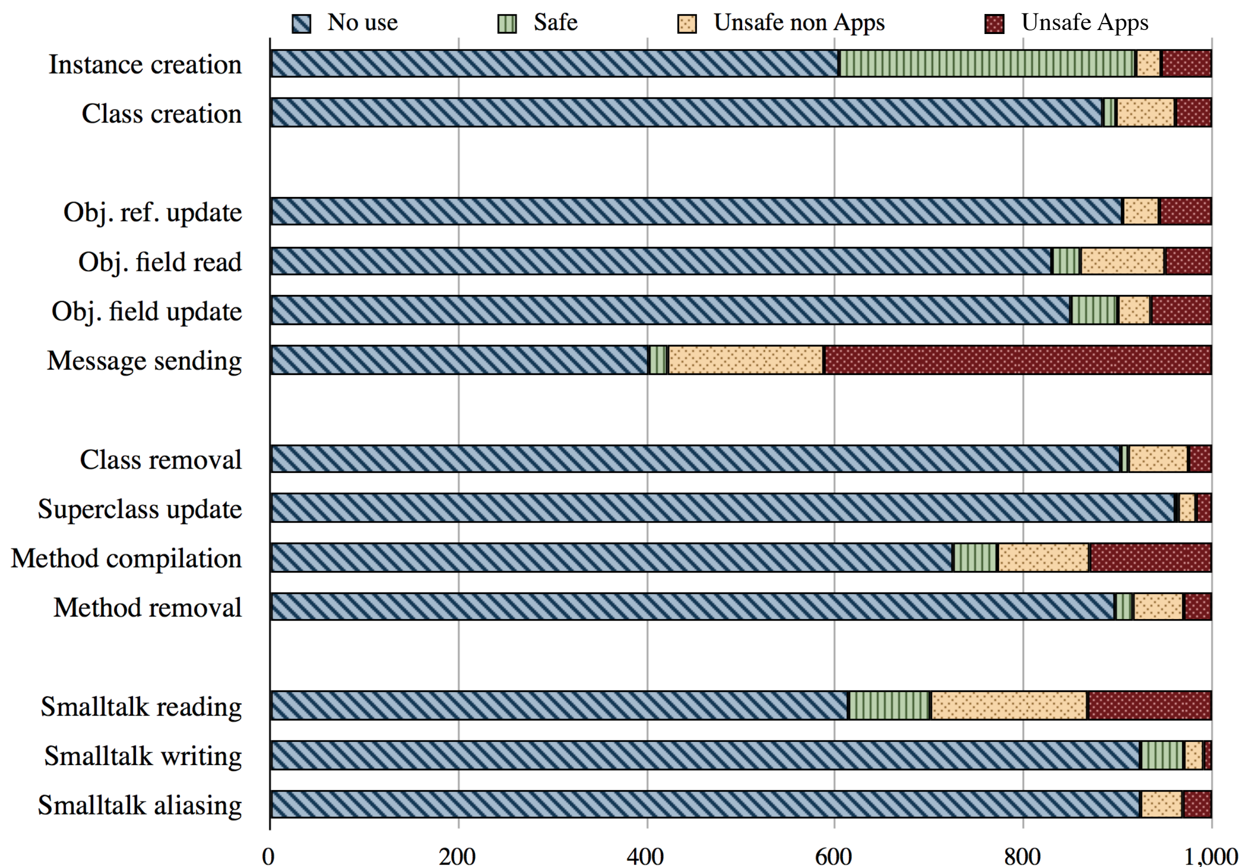

Figure 5.2: Per-feature distribution of all projects arranged by category of use.

- Finally, projects having only safe usages of a dynamic feature are a minority (excepting instance creation features), with an average of 22 projects.

The cases of instance creation, message sending and reading system dictionary are distinct: 40% of the projects make use of dynamic instance creation, but the majority of them only have safe usages; fewer than 10% of the projects use it unsafely. Message sending is even more widespread—60% of all projects use it—, but follows an opposite distribution of safe/unsafe usages: most of the projects use it in an unsafe fashion. In the case of reading the system dictionary: 39% of projects use this feature with a majority of unsafe usages, which represent 30% of all projects (half of them in applications). These three features are used pervasively

by all kinds of projects.

Interpretation

- The methods using dynamic features are a very small minority. However, the proportion of projects using dynamic features is larger, even if still a minority. This confirms hypothesis 1 that dynamic features are not used often, but shows that they cannot be safely ignored. An analysis of each feature is needed.

- Dynamic features are more often used in core system libraries, development tools, and tests, rather than in regular applications (hypothesis 2). However, it is important to remark that it is not the case that conventional applications use few dynamic features: applications gather nearly half of the unsafe uses. Considering that applications account for 83% of all analyzed projects, applications are nonetheless clearly under-represented in terms of dynamic feature usage compared to most other project categories (cf. Table 5.2).

- The three most pervasive features—Instance creation, Message sending and Reading the system dictionary—correspond to features that static languages such as Java support, confirming hypothesis 3.

5.4 How each dynamic feature is used

In this section, we analyze the usage of each dynamic feature in detail. For each feature, we distinguish between safe and unsafe usages as explained in Section 5.2.3. Additionally, we classify those usages between application code, and system, tools, language extensions and tests (Section 5.2.2). When they exist, we list common patterns of usage of the features. These results help us validate or invalidate hypothesis A4 (see page 67).

For each feature, we provide basic statistics (number of uses, number of unsafe uses, and number of unsafe uses in applications), and a bar chart showing the classification of each feature in various, feature-specific, patterns of usage. We

Dynamic Feature	%Safe	%Apps	%Tools	%Ext	%Syst	%Tests
Instance Creation	92.53	4.32 (0.69)	0.59 (1.25)	0.70 (1.70)	1.76 (**9.42**)	0.11 (0.61)
Class Creation	30	18.33 (0.31)	9.76 (2.21)	1.90 (0.49)	8.57 (**4.90**)	31.43 (**18.71**)
Object ref. update	0	45.66 (0.55)	12.22 (1.94)	6.75 (1.23)	24.44 (**9.78**)	10.93 (**4.55**)
Object field read	25.52	35.09 (0.56)	11.24 (2.40)	3.99 (0.97)	23.13 (**12.42**)	1.04 (0.58)
Object field update	61.14	18.67 (0.58)	5.20 (2.12)	1.80 (0.84)	12.91 (**13.29**)	0.28 (0.30)
Message sending	7.03	47.52 (0.61)	26.57 (**4.54**)	3.17 (0.62)	9.79 (**4.21**)	5.92 (2.65)
Class removal	6.24	10.91 (0.14)	8.05 (1.36)	1.56 (0.30)	10.91 (**4.65**)	62.34 (**27.70**)
Superclass update	7.89	42.11 (0.55)	18.42 (3.17)	7.02 (1.39)	7.02 (3.05)	17.54 (**7.93**)
Method compilation	60.02	18.25 (0.55)	7.10 (2.82)	3.22 (1.46)	6.32 (**6.32**)	5.10 (**5.32**)
Method removal	39.13	13.27 (0.26)	15.10 (3.94)	3.43 (1.02)	18.76 (**12.33**)	10.30 (**7.05**)
System dict. reading	48.52	15.86 (0.37)	11 (3.39)	2.47 (0.87)	11.36 (**8.83**)	10.79 (**8.73**)
System dict. writing	80.69	4.95 (0.31)	2.72 (2.24)	0.5 (0.47)	4.95 (**10.25**)	6.19 (**13.36**)
System dict. aliasing	0	28.02 (0.34)	27.54 (**4.37**)	4.83 (0.88)	15.46 (**6.18**)	24.15 (**10.06**)

Table 5.2: Per-feature distribution of safe and unsafe calls, where unsafe calls are sorted by project category. In bold: category that is considerably over-represented (over-representation factor > 4)

also provide percentage distributions among categories in Table 5.2. We highlight in bold categories that are particularly over-represented, measured by the over-representation factor (ORF). An ORF of 1 means that a category has a distribution of unsafe calls equal to its representation in the project corpus. The higher the ORF, the more over-represented are unsafe calls in a particular category. For instance, while only 2.5% of all projects belong to the System category, it is responsible for 23.56% of all unsafe instance creation occurrences (1.76% of all instance creations), hence the over-representation of unsafe instance creations in the System category is 9.42. Note that Application projects are under-represented for all dynamic features as denoted by an ORF smaller than 1; for Application projects the factor varies between 0.14 for the class removal and 0.69 for the instance creation feature.

First-class classes:

Instance Creation (2,732 calls, 204 unsafe, 118 in Apps). Figure 5.3 reveals that programmers use instance creation (basicNew) in a statically safe way, *i.e.* self, class name and super, (92.53%) while unsafe calls, *i.e.* variable and other, (see Table 5.2 for distribution) are restricted to a few occurrences (7.47%). Applications feature the most unsafe calls (118, *i.e.* 4.32%), but are actually under-represented as 83% of the projects are applications (ORF=0.69). On the contrary, System projects are the most over-represented (1.76%, ORF=9.42).

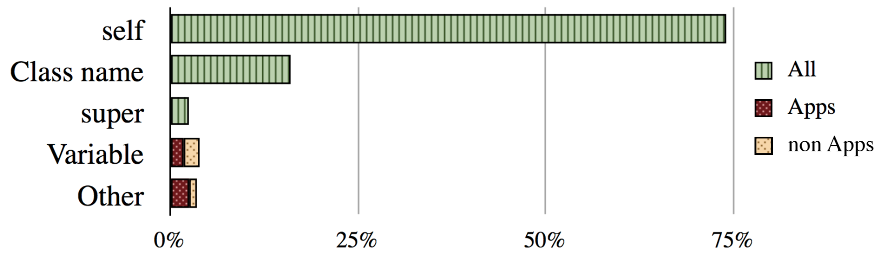

Figure 5.3: Safe/unsafe usages of instance creation.

The most common (and safe) pattern is self basicNew (74%): Programmers define constructor methods (as class methods) and inside them call basicNew. A common unsafe pattern (almost a third of unsafe calls) is to defer the choice of the class to instantiate via polymorphism (self factoryClass basicNew).

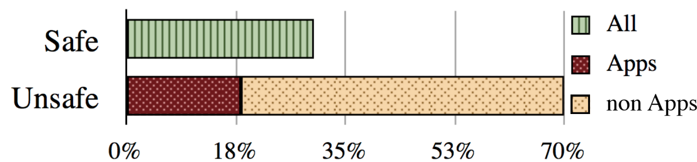

Figure 5.4: Safe/unsafe usages of class creation.

Class Creation (420 calls, 294 unsafe, 77 in Apps). Figure 5.4 and Table 5.2 show that a strong minority of cases are safe uses (30%); 18% of unsafe usages are in application (ORF=0.31), while more than 50% are in other project categories. Tests are extremely over-represented, with nearly a third of unsafe usages, with an ORF of 18.71. Indeed, tests often create temporary classes for testing purposes, and the ORF confirms that this practice is indeed very common. Likewise, System and—to a lesser extent—Tools are both over-represented (ORFs of 4.90 and 2.21, respectively), each having close to 10% of uses of the features; both project categories are infrastructural in nature and may need to create classes as part of their responsibilities. Most unsafe usages in Apps are in class factory methods generating a custom class name, such as:

```
FactoryClass>>customClassWithSuffix: aStringSuffix
    ↑ Object subclass: ('MySpaceName' , aStringSuffix) asSymbol.
```

To provide perspective, the code base we analyze contains 47,720 statically defined classes, showing that dynamic class creation clearly covers a minority of cases, less than 1%.

Interpretation

- Instance creation is the third-most used dynamic feature, but its usage is mostly safe, with only 118 unsafe usages in applications.
- The majority of class creation uses are unsafe, but most of those are located in non-application code, primarily testing code. A lot of unsafe usages appear to be related to class name generation.
- Some support is still needed for a correct handling of these features in static analysis tools. In particular, support for self-types is primordial to make usages of self and super tractable and hence safe.

Behavioral reflection:

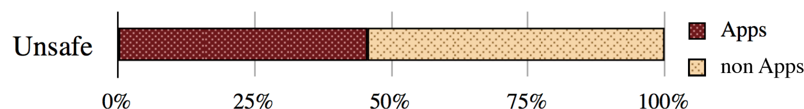

Figure 5.5: Unsafe uses of object references updates.

Object Reference Update (311 calls, 311 unsafe, 142 in Apps). According to Table 5.2, System projects particularly are over-represented (2.5% of projects account for 25% of calls, an ORF of 9.78). For instance, some low-level system operations need to migrate objects when their classes are redefined, and use become: for such a task. Applications however do use this feature somewhat extensively, with more than 45% of calls, see Figure 5.5. They are still under-represented (ORF=0.55).

Object Field Read (1254 calls, 934 unsafe, 440 in Apps). Reading object field values is a commonly used dynamic feature, accounting for 6.15% of all dynamic feature usages. The distribution of safe (only 25.5%) and unsafe usages is displayed in Figure 5.6. Unsafe usages can be further categorized either in calls using as argument (i) a variable (64.59%) or (ii) a complex Smalltalk expression (9.89%, referred to as Other). Most unsafe reading of object fields occurs in App projects (35.09%), while in System projects this feature accounts for 23.13% of all unsafe usages. This makes System projects extremely over-represented, as the ORF is 12.42. The other project categories are less represented, particularly Tests projects, which have nearly the same ORF as application (0.58 and 0.56, respectively). Most unsafe usages in category variable follow the pattern:

```
obj allInstVarNames do: [:ivar |
    (obj instVarNamed: ivar) doSomething]
```

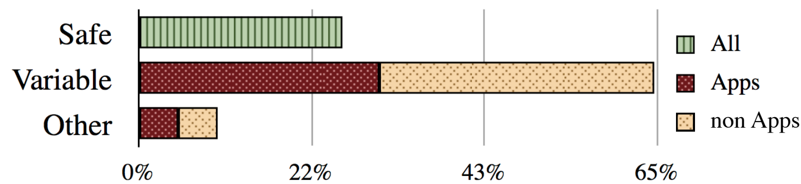

Figure 5.6: Safe/unsafe usages of object field reads.

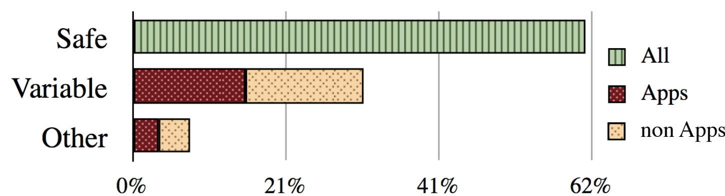

Figure 5.7: Safe/unsafe usages of object field updates.

Object Field Update (1,441 calls, 560 unsafe, 269 in Apps). Writing and updating the values of object fields is the fifth-most used feature. Figure 5.7

gives the distribution of safe (61.14%) and unsafe calls. Unsafe calls are split into: Variable (31.16%), when the first argument is a variable; and Other (7.7%), when the first argument is a compound Smalltalk expression, such as a method call. Unsafe calls in applications make up 18.67% of the total (Table 5.2), while unsafe calls in the System category make up 12.91% of all field updates (ORF=13.29), for reasons similar to the uses of object reference updates. Here again, Tests are under-represented, with an ORF of 0.30, and Language extensions have an ORF of 0.84.

The following pattern is extremely common (664 or 46% of all calls, with 398 calls in Applications):

```
MyClass basicNew instVarAt: idx1 put: value1 ;
              instVarAt: idx2 put: value2;
             ...
              instVarAt: idxN put: valueN.
```

This code snippet creates a new object and initializes all its fields with predetermined values. Smalltalk provides the storeString method, which serializes the object in the form of a valid Smalltalk expression that, when executed, recreates the object in the same state; it is a relatively common practice to save objects as executable expressions that way. It is actually surprising that Tests do not use this feature more heavily.

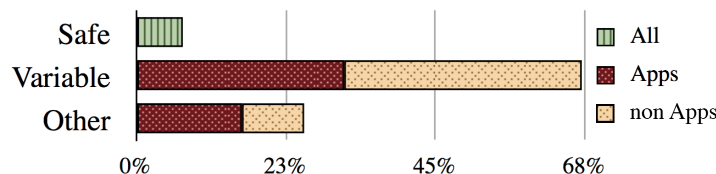

Figure 5.8: Safe/unsafe usages of message sending.

Message Sending (6,048 calls, 5,623 unsafe, 2,874 in Apps). This is the most used dynamic feature and accounts for 29.67% of all occurrences. Unfortunately, most of these usages (92,97%) are unsafe (Figure 5.8). This is not surprising: there is little value in calling a method reflexively if the message name

is constant. Two thirds of all calls use as argument a local variable, and more complex Smalltalk expressions are used in one fourth of cases.

Table 5.2 indicates that almost half of the message sending feature occurrences (47.62%) are unsafe calls inside App projects. If Apps are under-represented, as usual, the ORF of 0.61 is one of the highest. Tool projects follow with a quarter of all occurrences (26.57%, ORF=4.54); a possible explanation for that large over-representation is that tools often feature a UI, for which reflexive message sending is commonly used in Smalltalk—an example being the Morphic UI framework. System packages are also significantly over-represented (ORF=4.21), although the reason for that is less clear. The rest is split between the other project categories: Tests (5.92%) and Language-extensions (3.17%, under-represented).

Interpretation

- Supporting message sending is a priority: it constitutes almost 30% of dynamic feature usages; 60% of projects use it; nearly 93% of uses are unsafe. However, supporting message sending efficiently may be challenging. The state-of-the-art solution of Bodden *et al.* mixes enhanced static analysis with dynamic analysis to provide sufficient coverage [Bodden et al., 2011].

- The other three behavioral features—object reference update, and object field read and update—are used infrequently. The exception is System projects, which do use them pervasively: in all three, System projects have an over-representation factor in excess of 9.

- Object field updates are 60% safe, due to their usage as a serialization mechanism. This contrasts with field reads, whose safe usages are only 25%; field reads are mostly used with a dynamically-determined instance variable name. Reference updates are much more challenging to support.

Structural reflection:

Class Removal (385 calls, 361 unsafe, 42 in Apps). Class removal is one of the least-used features. According to Figure 5.9, safe usages are in the minority (6.24%); calls with a local variable as receiver make 80% of the calls; more complex

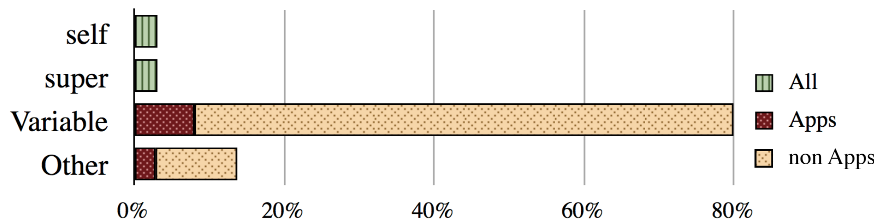

Figure 5.9: Safe (green)/unsafe(red and yellow) usages of class deletion.

calls make the rest. Unsafe usages in applications are also a minority (10.91% of usages according to Table 5.2, and an extremely low ORF of 0.14), whereas System projects have the same number of usages (translating to a much higher ORF of 4.65). Tests provide the overwhelming majority of unsafe usages with 62.34%. This massive over-representation (ORF=27.70, the largest by far) ties up with the heavy usage by tests of dynamic class creation (which also had a very high ORF. 18.71). A common pattern in tests (208 instances, more than 80%), is to create a new class, run tests on it, and then delete it.

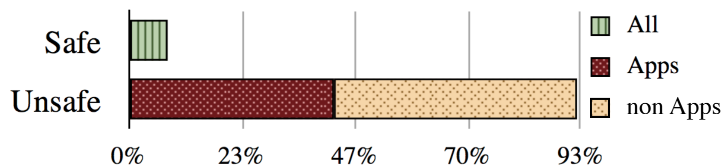

Figure 5.10: Safe/unsafe usages of superclass updates.

Superclass Update (114 calls, 105 unsafe, 48 in Apps). This feature is the least used with just 114 occurrences, 0.56% of all dynamic feature occurrences. As shown in Figure 5.10, safe calls account for 7.89% while 42.11% are unsafe calls inside App projects. Tests are the heaviest users (ORF=7.93, 17.54%), followed by Tools (3.17, 18.45%) and Systems (3.05, 7.02%). Since tests often build classes to run test code on, it stands to reason that they would also need to specify their superclasses.

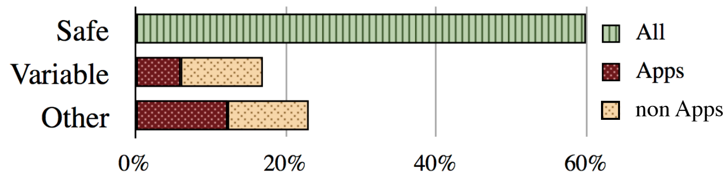

Figure 5.11: Safe/unsafe uses of method compilation.

Method Compilation (2,296 calls, 918 unsafe, 419 in Apps). Method compilation is the fourth-most used feature, with nearly 2,300 of the 14,184 calls. A majority of the usages (60%) are statically known strings, and are thus safe (Figure 5.11). Of the rest, 17% hold the source code in a variable, while 23% are more complex expressions— *i.e.* a string concatenation, which represents 40% of complex expressions.

Applications feature a bit less than half of the usages (18.25%, ORF=0.55), and are hence under-represented (but not less than usual); on the other hand, Systems (ORF=6.32), and Tests (ORF=5.32) are over-represented. This behavior is similar to the one in class creation (although less skewed towards Tests), and has similar reasons.

In addition, we manually classified the safe method compilations that are known statically in trivial methods (returning a constant or a simple expression), getter/setter (returning/setting an instance variable), and arbitrary code. We found that the vast majority (75.91%) of the methods compiled were trivial in nature, while getter and setters constituted 7.55%, with the remaining 16.55% being arbitrary. Examples of methods classified as "trivial" follow:

```
ClassA>>one
  ↑ 1.

ClassB>>equals: other
  ↑ self = other.

ClassC>>newObject
  ↑ self class new.
```

Note that the code base we analyze contains 652,990 methods, so we can hypothesize that the number of statically defined methods vastly outnumbers the

quantity of dynamically defined ones, but we cannot be sure of that fact without performing dynamic analysis.

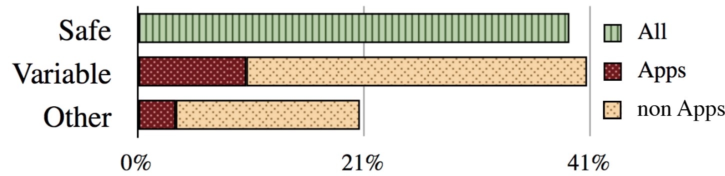

Figure 5.12: Safe/unsafe uses of method removal.

Method Removal (437 calls, 266 unsafe, 58 in Apps). Method removals are used relatively sparsely, and unsafe uses are much more prevalent in Tools, Systems, Language extensions, and Tests than in Apps, as shown in Figure 5.12. Safe calls make up 39.13% of all the calls; unsafe calls with a variable 40.73%; complex unsafe calls 20.14%. We see in Table 5.2 that Apps are clearly under-represented (a low ORF of 0.26): System projects on the other hand are very over-represented (ORF=12.33), as are Tests, to a lesser extent (ORF=7.05); Tools follow (ORF=3.94). The low ORF in Apps is similar to Class Removal.

Interpretation.

- Besides method compilation, structural reflective features are rarely used. In addition, the vast majority of application projects do not use these features. It appears that support for superclass update, class removal, and method removal does not need to be as urgent/efficient than other features.

- Class removal seems to be quite correlated with class creation, which is expected. Table 5.2 shows that all project categories show similar numbers of usages (with Apps creating more classes than they remove); the total number of calls are also similar (385 vs 420). The over-representation factors are also quite similar.

- Changes to methods (method compilation and removal) have a large proportion of safe usages (40 to 60%). However, this significant proportion of unsafe uses means that support for method compilation cannot be neglected in the design of static analysis tools.

- Tests are the largest users of structural reflection, as they are heavily represented in all features. System projects follow (2 out of 4).

System dictionary:

Reading (4338 calls, 2233 unsafe, 688 in Apps). Reading the system dictionary is the second most used dynamic feature and accounts for 21.41% of all usages. Approximately half of usages are unsafe (51.5%, see Figure 5.13). Most unsafe usages occur by passing a local variable as argument to read objects stored in the dictionary (24.55%); more complex Smalltalk expressions are used in 26.92% of all unsafe usages.

As indicated in Table 5.2 the system dictionary is mostly read in App projects (15.9%), followed by System (11.36%), Tools projects (11.00%), and Tests (10.79%). Given the relative size of these projects categories, reading the system dictionary is overly represented in these categories compared to App projects (ORF=0.37). The trend we have been seeing earlier, with System and Tests often being heavy users, is confirmed (ORF=8.83 and 8.73, respectively)

We found two common patterns: (1) using the system dictionary to check the existence of a class (60% in safe usages), and (2) accessing the class reference through the system dictionary (30% in safe usages). Below are some examples of the previous pattern:

```
Smalltalk at: #MyClass
    ifPresent: [...do something...]
    ifAbsent: [...do something else...] .

myClassRef := Smalltalk at: #MyClass.
```

These patterns are not the monopoly of safe usages, but are also present in unsafe usages in similar proportions.

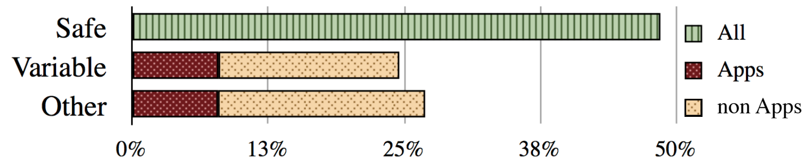

Figure 5.13: Safe/unsafe uses of Smalltalk readings.

Writing (404 calls, 78 unsafe, 20 in Apps). Writing in the system dictionary is much less common than reading from it. By accounting for 1.99% of all dynamic feature usages it is one of the less-used features. Around a fifth of its usages are considered to be unsafe (19.3%). Similar to previous features, unsafe usages are split in two groups: passing a local variable to the dictionary to identify the object to be written (12.87%); and passing a complex Smalltalk expression to the dictionary (6.44%)(cf. Figure 5.14).

Unsafe writing to the system dictionary mostly occurs in Tests (6.19%), subsequently followed by Apps and System projects, both with a share of 4.95%. Both System projects and Tests are overly writing to the system dictionary compared to the relative sizes of these project categories (ORF=10.25 and 13.36, respectively).

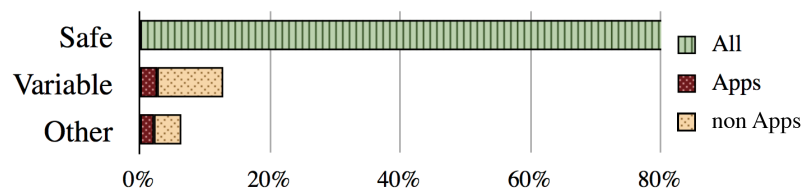

Figure 5.14: Safe/unsafe uses of Smalltalk writings.

Aliasing (207 calls, 207 unsafe, 58 in Apps). Aliasing the system dictionary is even less common than writing to it, making it the second least used of all analyzed dynamic features behind superclass update, only accounting for 1.02% of all dynamic feature usages. However, as mentioned in Section 5.2.3, we consider all occurrences of aliasing to be unsafe. As Figure 5.15 illustrates, almost all usages

of system dictionary aliasing happen by passing the dictionary as parameter to a method (89.86%); 7.7% are local aliases, which appear superfluous, and 2.4% are aliases to fields.

Most occurrences of system dictionary aliasing are in App projects (28.02%), followed by Tools (27.54%), Tests (24.15%) and System projects (15.46%). Hence, as with most other dynamic features, Tests (ORF=10.06) and System (ORF=6.18) projects are over-represented (and Tools, to a lesser extent).

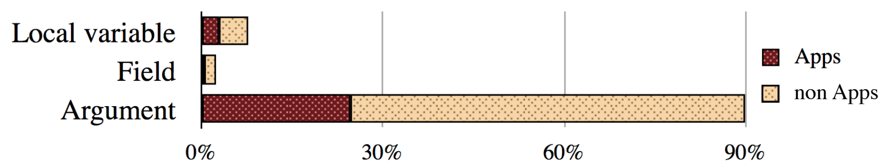

Figure 5.15: Safe/unsafe uses of Smalltalk aliasing.

Interpretation.

- Smalltalk reading is the second most used dynamic feature with 21% of all usages. Testing for the existence of a class or obtaining a reification of it is also supported by Java, confirming assumption 3.
- Smalltalk writing is used infrequently. However more than 80% are safe usages. Tests and System applications are over represented (see Table 5.2), with 6.19% and 4.95% respectively.
- Smalltalk aliasing is the second least used dynamic feature. A few usages (less than 10%) are direct aliasing to local variables, which could be avoided through straightforward substitution. Other usages would require inter-procedural analysis to track down how the dictionary is used afterwards.
- Tests and System projects are largely over-represented for every feature.

5.5 Discussion

We discuss whether each of the assumptions and research questions we mentioned in the introduction is valid or not, and provide guidelines for each feature we studied.

1. **Dynamic features are rarely used.** Dynamic features were found to be used in a small minority of methods—1.76%. Assumption 1 is validated.

2. **Dynamic features are used in specific kinds of projects.** We conjectured that core system libraries, development tools, language extensions, and tests, were the main users of dynamic features. If these categories use on average much more dynamic features than regular applications (the 17% of projects make 56.57% of unsafe usages), the latter still makes up nearly half of all the unsafe usages. Going further, a statistical test showed us that there were more usages of dynamic features in projects not classified as applications but we found the effect size to be small. As such, assumption 2 is validated, albeit with a smaller effect than expected. If we look at categories individually, the over-representation factors of Table 5.2 show us that Systems and Tests are by far the largest users of unsafe dynamic features, Tools are also over-represented, whereas Language Extensions are close to normal, and Applications are systematically under-represented.

3. **The most used dynamic features are supported by most static languages.** The three most used features, reflective message sending, reading the system dictionary and instance creation, are supported by the Java reflection API, validating assumption 3.

4. **Some usages of dynamic features are statically tractable.** We found that 4 features (instance creation, object field updates, method compilation, and Smalltalk writing) have a majority of safe uses. Three others (object field reads, class creation and method removal) have a strong minority (more than 30%) of safe uses. Assumption 4 is validated.

Even if dynamic features are used in a minority of methods (1.76%, validating assumption 1), they cannot be safely ignored: a large number of projects make use of some of the features in a potentially unsafe manner. We review each feature on a case-by-case basis, and in order of importance (as determined by overall usage of the features).

Message sending is the most used feature overall, with 60% of projects using it and a majority of unsafe uses. Supporting it is both challenging and critical.

Smalltalk reading represents the second most used feature, with almost 40% of projects using it. Half of system dictionary reads are unsafe. Supporting it is crucial and also challenging.

Instance creation is used by 40% of the projects, but can be considered mostly safe if a notion of *self types* is introduced, as in Strongtalk [Bracha and Griswold, 1993].

Method compilation is used in an unsafe manner by a little over 20% of the projects, and as such also needs improved support.

Object field reads and updates are the last of the features that has a somewhat widespread usage. Although reads usages are mostly unsafe, updates are mainly safe.

Class creation and removal are heavily used in tests, but class creation is used in applications as well.

Smalltalk writing is rarely used, less than 8% of projects used it. Moreover, four of five system dictionary updates are safe.

Object reference updates are somewhat problematic, as nearly 45% of the usages are in applications. Supporting such a dynamic feature is also a challenge.

Method removal has a large number of safe uses, and is primarily used outside applications.

Smalltalk aliasing are problematic, although occasionally used (7.9% projects). Supporting it is not vital, but programmers must take them into account.

Superclass updates is a somewhat exotic feature whose usages are few and far between.

As a rule of thumb, we conclude that message sending, system dictionary reads, method compilation, instance creation, object field reads and updates, and to a lesser degree also class creation, system dictionary updates, and object reference updates, are particularly important dynamic features that static analysis tools, type systems, and compiler optimizations should support well. Of less importance are class and method removals, Smalltalk aliasing and superclass update since they are rarely used in an unsafe manner in application projects, nonetheless language designers cannot afford to completely ignore them.

5.6 Why do developers resort to using dynamic features? (and what to do about it)

Beyond a state of the practice on the usage of dynamic features, we also wish to understand why developers end up using them. Even if Smalltalk is a language where these features are comparatively easier to access than most programming languages, developers should only use them when they have no viable alternatives, as they significantly obfuscate the control flow of the program, and add implicit dependencies between program entities that are hard to track (*e.g.* a dynamic invocation of a method does not show up in the list of users of the methods). As such, any usage of a dynamic feature that is superfluous, or that can be removed at a moderate cost, should be removed, or at least carefully considered for removal.

In this section, we answer the two following research questions:

RQ1: What are the principal reasons why developers use dynamic features?

RQ2: Are certain types of dynamic feature usages refactorable? Can they be removed, or can unsafe usages be refactored to safe ones?

However, these questions cannot be addressed by a large-scale quantitative analysis as we performed above; each dynamic feature usage needs to be manually inspected to determine the reason of its existence, and whether it can be removed. Thus, we had to reduce the scope of the study.

5.6.1 Methodology

Sample size From a total of 1,000 Smalltalk projects, we collected 20,387 occurrences of dynamic features in Section 5.3. These occurrences represent our initial data set. Due to the fact that manual inspection is required, we extract a representative sample set that we inspected further. We establish the sample set size (n) with the formula [Triola, 2006]:

$$n = \frac{N \cdot \hat{p}\hat{q} \cdot (z_{\alpha/2})^2}{(N-1) \cdot E^2 + \hat{p}\hat{q} \cdot (z_{\alpha/2})^2}$$

We keep the standard confidence level of 95% ($z_{\alpha/2}$) and the standard 5% margin of error (E). The proportions source code that is refactorable ($\hat{p}$), and of source code that is not refactorable ($\hat{q}$) are unknown a priori, so we consider the worst case scenario ($\hat{p} = \hat{q} = 0.5$, *i.e.* $\hat{p}\hat{q} = 0.25$). Statistically speaking, the population of our data set (20,387 dynamic features usages) is relatively small. Because of this, we include the finite population correction factor directly into the formula (N).

The sample size computed from the formula is 377. As we do not have previous information about the distribution of refactorable source code, we employ random sampling without replacement to select a reliable sample set from our initial data set. Table 5.3 shows the obtained distribution by feature. The first column is the name of each analyzed feature; the second represents the data set size; and finally, the third shows the sample size.

Inspecting dynamic feature usages In order to understand the rationale for using a given dynamic feature, and to determine whether and how a given feature is

Dynamic Feature	**Data size**	**Sample size**
Instance Creation	2,732	58
Class Creation	420	10
Object ref. update	311	5
Object field read	1,254	23
Object field update	1,441	14
Message sending	6,048	125
Class removal	385	8
Superclass update	114	5
Method compilation	2,296	34
Method removal	311	3
Smalltalk reading	4,338	79
Smalltalk writing	404	8
Smalltalk aliasing	207	5
Total	20,387	377

Table 5.3: Per-feature distribution of the sample set size

refactorable, we employed partial program comprehension techniques ([Erdös and Sneed, 1998]). Our infrastructure allows us to browse the source code of a dynamic feature usage, and, when necessary, browse the source code of the project using it, as well as inspecting the list of methods calling the one where the dynamic feature is used. As such, each code fragment was inspected for as long as it was deemed necessary in order to understand the reasons behind the usage of the dynamic feature, and whether it was refactorable. The author of this thesis work and his advisors inspected the sample; each source code element was inspected by at least two authors.

To abstract away from the raw information in the sample, we classify the kinds of dynamic feature usages into several categories. We performed several iteration steps where the authors would examine individual examples in the sample, and either assign it to a category, or create a new one. After each step, we discussed the classification and altered the definition of the categories. This was performed until we arrived to an agreement on a stable classification. The classification was stored in an online repository, which was used as support for the discussion. We describe the different categories in Section 5.6.2 below.

In addition to classifying dynamic feature usage rationales, we also report,

though less systematically, on the refactorability of dynamic feature usages. In particular, we pay attention to the locality of refactoring when possible. This is because refactorings that are local to a given method body are much easier to perform than global ones, such as modifying all call sites of a given method.

Finally, we also looked at prominent application domains where dynamic features are used, as reported in Section 5.6.3. We mention these when a particular kind of application is overwhelmingly represented in a given category.

5.6.2 Categorizing user intention when using dynamic features

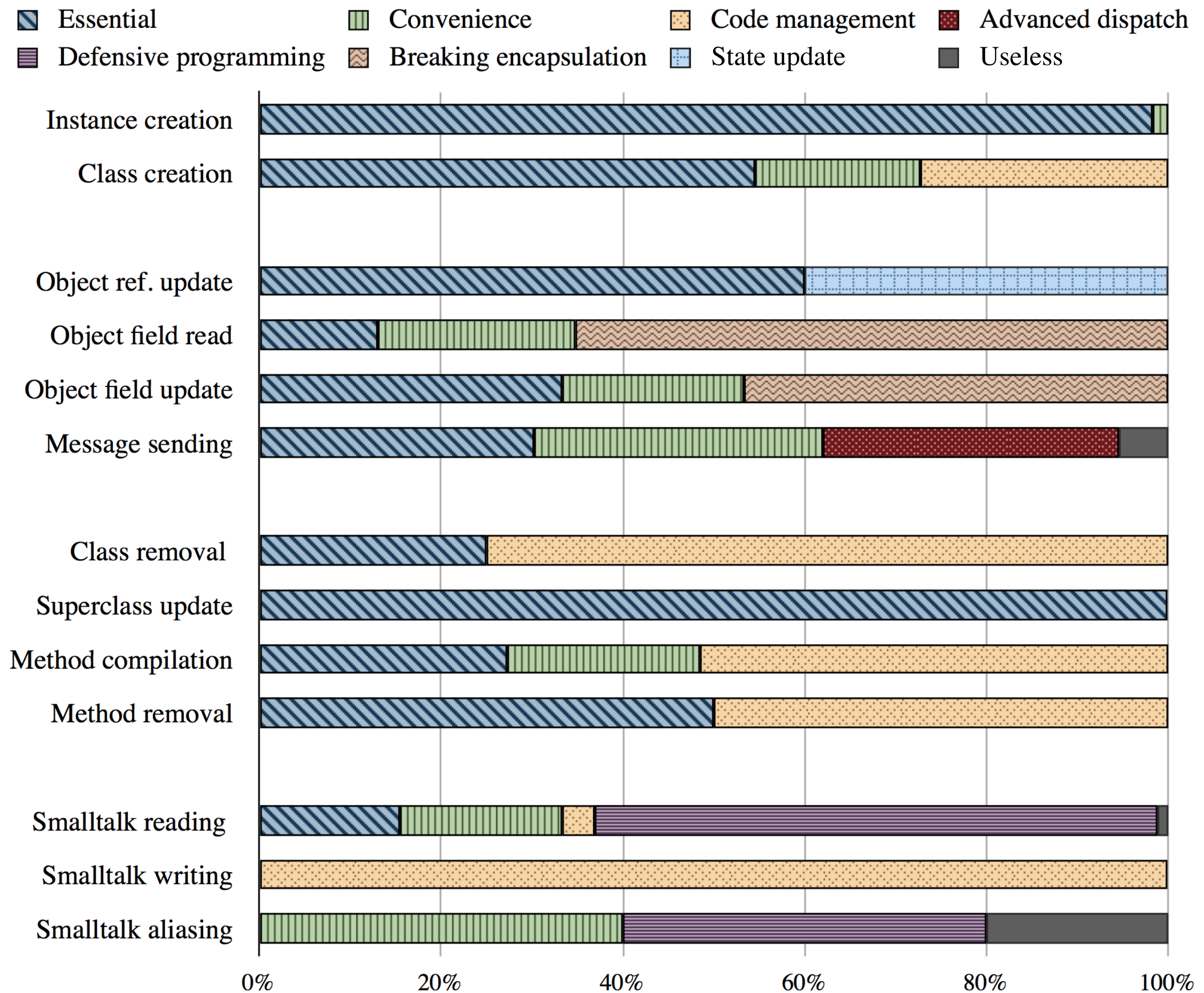

Figure 5.16: Per-feature distribution of user intention.

In this section, we describe the eight categories we found in the inspection of the sample set, mining for user intent. We start with the most generic categories, that are significantly present in several features. We then introduce categories that are related to some specific features. We explain each category, describe

some interesting cases extracted from the samples, and discuss the potential for refactoring of the different scenarios. We end this section with a brief discussion of the false positives detected in our analysis.

Figure 5.16 summarizes our results, showing the per-feature distribution of user intention categories. For each feature, we count relative usages (percentage) discarding false positives.

Essential usages. Usages of dynamic features classified as *essentials* are intrisic usages of the dynamic feature. They constitute 37.5% of all dynamic feature usages—more than a third. All but two categories of dynamic features (Smalltalk writing and aliasing) have essential usages.

For instance, a remote communication framework needs to perform dynamic method invocation based on the instructions received from a socket. Hence, the original functionality of dynamic method invocation is embedded in a *wrapper* that refines it to specify a communication protocol, but still uses the original functionality deep down. This category is present in almost all features.

In our analysis, we found that programmers wrap essential dynamic feature usage within code of their own to alter their behavior, *e.g.* adding pre/post-conditions, or providing extended behavior such as the remove communication example mentioned above. Additional examples include essential dynamic feature usages aimed at facilitating debugging (logging), and wrappers around object field accesses in order to mirror changes made to an object in an object database. Language extensions (*e.g.* a dynamic component model on top of Smalltalk objects) also make use of dynamic features in such a fundamental manner.

In general, these usages cannot be refactored, because the possibilities to take into account are very large, even potentially infinite (*e.g.* they could depend on user input). Even when the usages may be refactored theoretically, in many cases it is at a prohibitive cost. For instance, removing a wrapper around an object field access in an object database could be achieved only at the cost of modifying every access to the state of every object susceptible to be stored in the database. Likewise, reflective method invocation in a remote communication framework could be replaced by a mechanism similar to Java RMI, where remote interfaces are declared statically and proxies are generated by the compiler. Proposals to alleviate

the burden of RMI programming in Java are, unsurprisingly, based on reflective invocation.

Occasionally, some of these instances can be refactored. A particular example we found was a wrapper used in debugging that executed a given method, but only if it belonged in a predefined (and small) set of allowed methods. Since the number of possibilities was small (only 4), the cases could be enumerated and the dynamic feature usage removed.

Beyond this wrapping behavior, we found two other categories of essential usages. The first is the overwhelming majority of dynamic class instantiation (98.3%—except in one case). These are essential because they are the very means by which objects are created in the language; however, as we mentioned previously, self types [Bracha and Griswold, 1993] can be used to make these usages safe.

The second one is test code that explicitly tests dynamic features. Obviously, removing it would negatively impact the test coverage of the projects, so these usages can not be parted with either, nor can they be refactored away.

Convenience. Reflective features are powerful, providing flexibility to programmers that can at times be abused. Since Smalltalk makes it extremely easy to call a method based on its selector (a symbol), iterate over sets of methods or instance variables, programmers often use these features to shorten repetitive source code, sometimes at the expense of readability. Convenience usages are pervasive: they are the second-most important category, just shy of 20% of the usages of dynamic features; 8 out of 13 dynamic features are used as such; of note, 32% of dynamic method invocations are classified as convenience.

An example is the following, which chooses a message to send based on a condition:

```
classesSelected := classesSelected
    perform: (aBoolean ifTrue: [ #copyWith: ] ifFalse: [ #copyWithout: ])
    with: (self classes at: anInteger ifAbsent: [ ↑ self ]).
```

This could be rewritten as the slightly longer, but far more legible:

```
arg := self classes at: anInteger ifAbsent: [ ↑ self ].
classesSelected := aBoolean
    ifTrue: [classesSelected copyWith: arg]
    ifFalse: [classesSelected copyWithout: arg].
```

There are surprisingly many of these types of usages in our sample, including a similar one where a class is similarly chosen based on a boolean condition, and is then instantiated.

Other examples include specifying a list of methods in an array, and then iterating over it, executing a method at each step of the iteration. We found several test cases that are organized in this fashion, where common behavior before and after the test is executed in the body of the loop. Some programs make use of the fact that Smalltalk methods are classified in categories, in order to execute all the methods belonging to a certain category. A particular example of this scenario executes all the methods in a given category, and returns the list of the returned values. As it turns out, each method actually returns a constant string, making the set of dynamically-executed methods equivalent to returning an array of strings (with, among others, the added benefit that an array of string is immune to errors in classifying methods in the wrong category). Another example implements an "undo" mechanism by setting up an array of method names of undo methods. To undo an action passed as parameter, it computes the index of the undo action and retrieves the associated method in the array, to execute it. Of course, such a scheme is sensitive to the order of the methods in the array.

The same type of behavior is also common in order to iterate over instance variables names, or classes names. Unlike the wrapper case, the majority of these usages feature a small number of possibilities, and are possible to refactor locally, by simply enumerating the handful of possibilities. A minority are a bit more challenging to refactor, as they involve scattered modification (*e.g.* converting a set of dynamically-invoked methods to an array of strings).

Dynamic code management. This category deals with the runtime creation of new code entities (classes and methods), their modifications, and their disposal. Smalltalk provides free access to the compiler as a class of the system, in the same manner as many dynamic languages provide an "eval" function (*e.g.* Javascript [Richards et al., 2011]). As such creating and deleting classes and methods is common, as we have shown above. Dynamic code management is the third most widespread category in terms of versatility, with 6 out of 13

dynamic features. Overall, 8.6% of all usages belong to dynamic code management, including all of Smalltak writing, and the majority of class removals, and of method compilations and removals.

The reasons for this are multiple:

- Some tests need to create classes and methods as part of their fixtures, and delete them afterwards.
- Some system projects implement prototypes (as in Javascript), by defining one class for each would-be prototype.
- Others generate basic or more elaborate methods from parameters they are given. In our sample we found several implementations of auto-generated getters and setters for instance variables; needless to say, a standard implementation for this would be useful. Other cases are much more elaborate, involving iterations over a set variables to either generate several methods, or several statements in a method.
- A lot of the code generated in fixtures for test cases consist of constant methods. We found other uses for constant methods, namely patching the code of another application in order to fix a known issue. This is of course a brittle strategy, as it ignores the possible changes in the other application.
- Yet another use of code generation is as an ad-hoc serialization mechanism. Using structural reflection, each Smalltalk object can generate a piece of code that, when executed, returns a copy of the object itself. This produces sequences of calls to reflective field accesses, each of them setting up one field of the object. This expression can then be compiled in a method of a class in order to re-create the object at anytime, and is used in test fixtures.

Cleanup code—removing code entities—is also somewhat prevalent (especially, as we remarked above, in test code to remove previously generated classes or methods). Other cleanup operations remove all the subclasses of a given class, which is known to contain generated code. A variant of this is the dynamic recompilation of methods, replacing the statements that they contain with an empty method body. We found this in a handful of cases, when a program dynamically changes from

development mode to deployment mode: methods logging behavior are altered to do nothing instead. Of course, a similar behavior could be achieved in a program that keeps its development/deployment mode more explicitly as a boolean status, which is checked by the methods above. Barring this particular case, and replacing ad-hoc serialization with a more robust serialization mechanism, refactoring these is difficult.

Advanced method dispatch. A large part of the usages of dynamic method invocation are alternative method dispatch mechanisms—32.6%, or 11% of all the dynamic feature usages. We found the following:

- An object stores another object and an associated method name, for later invocation. This is very common in UI frameworks, where the method is expected to be called when a UI action takes place. This is a lightweight variant of the Observer and Listener design patterns. The standard UI framework available for Pharo, Morphic, provides a very simple way to set up UI notifications as follows:

  ```
  button on: #eventName send: #methodToExecute to: receiver
  ```

  In this case, two symbols are passed as argument, one being the name of the event to react to (such as button click, double click, etc.), and the other the name of the method to call on the third argument. If the event handler determines that the message needs an argument, it will also provide the source of the event as an argument. As a side note, this example shows that passing method names through symbols is problematic, as in this example, one symbol is a method name, while the second is not.

- An object performs a dispatch on itself based on a method name it receives as argument, or a method name that is obtained by processing the argument. This is again a lightweight variant of other design patterns, *e.g.* the Visitor or Strategy patterns. Examples that we found are an interpreter class, which interprets a stream of bytecodes represented as symbols, and dynamically invokes the method bearing the same name as the bytecode. Other examples

are more complex, where the method name is constructed from one parameter (adding a prefix or suffix to match the name of the method), or two parameters (implementing an ad-hoc double-dispatch mechanism).

Of course these are quite fragile, as they are very sensitive to renaming: only a runtime error will give an indication that a method has changed, or that a new type of object needs to be handled. The more heavyweight versions of these patterns are not sensitive to these errors occasioned by renaming, but may trigger an explosion of small "glue classes", such as the anonymous Java listener classes that are omnipresent in Java UI code. Barring additional support, such a refactoring is not trivial.

The well-known lightweight solution for these programming idioms is to use first-class functions, as found in languages such as Lisp, ML, Haskell, or Scala. First-class functions can be type checked, while method names as symbols carry no type information. Surprisingly, this is also possible in Smalltalk, because Smalltalk code blocks are essentially anonymous functions! However, their syntax makes them a bit more verbose than symbols, which make programmer opt for the more concise alternative in many cases. A straightforward iteration on a collection is achieved by:

```
collection do: [:eachInteger | eachInteger squared]
```

But many Smalltalk dialects also support the more concise:

```
collection do: #squared
```

Since dynamic message sending achieved by passing names of methods is the most used feature in our corpus, it seems worthwhile to explore more robust mechanisms than plain symbols. Such a refactoring would be tedious, but not complicated.

Defensive programming. Due to the nature of Smalltalk as a dynamic language that, in addition, relies on whole system images, one is never sure if the environment contains the classes necessary for the correct execution of a program. Recall that accessing a class is done via the system dictionary; if the class is not

defined, a null pointer will be returned. Defensive programming is distributed accross two features only (Smalltalk reading and aliasing) but is prevalent in these (61.9% of reading; 40% of aliasings; 14.17% overall).

Most code just assumes that all classes are present, but some code actually checks in the system dictionary if the required classes exist. Upon discovering that a needed classes does not exist, we found programs that react differently: some report the error to the user; some return a default value computed without the missing class; some use an alternative class instead; and finally, some dynamically download and install a given version of the package that contains the class. Introspecting the system dictionary cannot be refactored in the context of a dynamic environment like Smalltalk; a better way to solve the problem would be to improve the way Smalltalk handles package dependencies, either at the language or infrastructural level.

Breaking encapsulation. Reflective field accesses are sometimes used to circumvent encapsulation, because the API of an object does not permit access to its fields. They constitute 5.77% of all the usages, but 65.3% of object field reads, and 46.7% of object field updates.

Breaking encapsulation is common when a test needs access to a field but the programmer does not wish to expose the field for the rest of the world. Other instances of this usage exist outside of tests, for which the only solution would be to extend the API to permit access to such a field. Another solution would be to extend the language to better support privileged access to the internals of an object. Recall that Smalltalk is strongly encapsulated in the sense that fields are not accessible from outside an object. On the other hand, all methods are. Java or C++ support different visibility mechanisms used to control encapsulation at the level of both fields and methods. Even with such mechanisms, reflective access is needed at times, because they are not very flexible. Encapsulation policies are a flexible alternative to address this issue [Schärli et al., 2004].

We found one instance where a specific field was accessed reflectively, despite the presence of the accessor in the source code. In fact, the accessor was also performing additional computations, that the calling code did not wish to take

place. A source code comment stated: "Ugly, but fast", indicating that the user was perfectly aware of the problem, but chose to ignore it on purpose.

Most examples we found break encapsulation in a systematic manner, iterating over all the fields of an object, sometimes recursively. This is the case of generic object copiers, and of some persistence frameworks that need to write objects on disk. It is also the case of the ad-hoc serialization mechanism mentioned above, which generates source code that, when evaluated, creates a copy of the object. These would be prohibitive to refactor, as every object would need to implement its own version of copy or serialization methods.

State update. Since object reference updates is a seldom-used feature, the corpus that we inspected does not feature a lot of these usages—only 5. We think that it is worth mentioning that out of these 5, 2 made the receiving object itself change its class before resuming operation.

It is interesting that this kind of usage of reference update directly corresponds to *state update* in typestate-oriented programming languages [Aldrich et al., 2009]. Such languages support stateful resources with state-specific behavior and representation. As such they are the language-level equivalent of the State design pattern. Tracking down such state changes statically is not trivial, but is feasible, even in a gradual manner [Wolff et al., 2011].

Useless. A small, but still noticeable categories of usages (2.36%) were truly useless usages of dynamic features, where the usage did not appear to have any kind of benefit over its non-dynamic counterpart.

For instance:

```
(html effect id: id; perform: #appear)
```

Is strictly equivalent to:

```
(html effect id: id; appear)
```

We conjecture that these usages slipped in when a programmer reused and adapted a code fragment found somewhere else, and did not notice the dynamic usage was unnecessary. All of these can be safely replaced with their non-dynamic equivalent.

False positives. The final category we found were false positives, demonstrating the limit of our previous automated analysis. We fortunately only found barely more than a handful of them:

- We found seven usages of the Smalltalk system dictionary as a mean to define and alter global variables instead of storing classes. When programmers need to share state across large portions of their programs, they usually instantiate the Singleton design pattern, but the Smalltalk dictionary offers a "quick and dirty" alternative.
- One usage of the method removeFromSystem: (class removal) was calling a similarly named method that had a wholly different purpose. It appears that the original method name is not specific enough and can be overloaded.
- One usage of the method superclass: was calling a method with a similar intent, but which did not end up using a dynamic feature; this was a call to an object that mirrors an actual Smalltalk class—as part of a type inference system—but that does not impact the actual class.
- Finally, one call to compile: was found in "junk code"—obviously incorrect code that was never meant to be executed, but part of a fixture for a test case. Often, these fixtures are compiled dynamically and removed when the test finishes, but this one was statically defined—in a method named "foo".

This totals 10 false positives, or 2.65% of the sample (recall that the percentages reported above exclude false positives). This low figure gives us strong confidence in the results of the preceding section.

5.6.3 Types of applications

In the sample we inspected, we noticed that some types of applications had a particular frequency of usage of dynamic features, as well as a particular distribution of usages.

Testing. As we already hinted at earlier, unit tests constitute a large proportion of all the usages of dynamic features. Unit tests have several reasons to be heavy users of dynamic features:

- Testing the dynamic features themselves. In order to achieve a good code coverage, all the functionality of a system should be tested, including the dynamic features.
- Generation of test objects. In some cases, tests need to generate classes and methods as part of their behavior. In addition, these objects need to disposed of once testing is over. The easy access to the compiler makes it trivial to do so, making this a very common occurence.
- Bypassing the public API. White-box testing is done with the knowledge of how the code under test works. A test may need to set up an object in a way that should not be possible in the public API, such as accessing a field that should not be accessed by regular clients. In these cases, several test cases access the instance variable by name, or by index. In other cases, objects that are fixtures of test cases are saved in source code, and restored by accessing their fields directly

All these factors make it often difficult to refactor test cases so that they do not use these features. It can be argued that this is not as problematic, as test cases are not part of the core of the application. However, the maintainability problems encountered while co-evolving application and test code are not considered in this study; further analysis is required to shed light on this subject.

Other types of applications. We noticed 3 other types of applications that were prominent, and with specific usage patterns.

- UI applications make heavy usage of dynamic method invocation as a lightweight form of an event notification system; the usage of this idiom is so pervasive that it has spread to other types of event handling systems, such as the Announcements framework for Pharo.

- Several frameworks that communicate with databases, or implement object databases, make heavy usage of serialization and de-serialization of objects. In order to do so in a generic way, they reflect on the structure of the objects they need to serialize, using field access reads and field access writes.

- Low-level system support code uses object field reads and writes to implement copy operations, saving the state of the system to disk, and convert numbers and strings from objects to compact bit representation.

5.6.4 Summary

We found a variety of reasons why developers use dynamic features, in a spectrum ranging from essential, fully-justifiable reasons, to ones that can, for all intent and purposes, be considered useless.

- Some of these dynamic feature usages are unavoidable, as they are a direct mapping to the dynamic feature itself (such as method dispatch in a remote communication framework, or tests of the dynamic feature themselves). Removing or refactoring these usages is most of the time impossible.

- Other usages point at limitations in the programming language (lack of first-class methods, privileged access to the private attributes of an object, objects changing state). Beyond extending the language, other solutions to most of these problems exist in the form of design patterns. Refactoring these would often be costly, however, and would trade one kind of complexity with another.

- Yet another class of usage deals with the generation and removal of new source code entities. There is no clear guidelines to address these cases, however.

- Defensive programming code idioms are in the majority spawned from the nature of Smalltalk as an image-based programming system, with historically little support for handling dependencies between packages.

- Finally, the mere availability of such powerful dynamic features make programmers abuse them in the name of conciseness. To save a few lines of code, programmers will go to considerable lengths, at times producing code barely shorter, but much harder to understand—dubious gains, to say the least. Fortunately, most of these usages can be easily removed as they are not critical to the program.

5.7 Threats to validity

Threats to construct validity. We classified the projects into categories in order to investigate whether certain categories use dynamic features more often. We may have misclassified some of the projects. However, the author of this thesis work and his advisors individually classified all projects and discussed classification differences before coming to an agreement for each project.

Our list of methods triggering dynamic features is not exhaustive. Our criterion for inclusion of a given method was whether it was "standard", *i.e.* part of the Smalltalk-80 standard API. Non-standard methods triggering dynamic features were left out, however their usage is limited (for instance, there are 64 usages of the ClassBuilder class instead of the subclass: selector, and only 7 in regular applications).

A risk we had was to perform an analysis on a corpus that contained a high proportion of false positives. While performing our manual analysis of the sample, we kept track of the number of false positives we found, in order to gather an estimate of the false positive in our corpus. We found 10 false positives in our sample of 377 dynamic feature usages, or 2.65%, which we judge to be acceptably low for our general findings to hold.

We only use static analysis as it would be impractical to perform dynamic analysis on 1,000 projects. If costly, dynamic analysis would allow us to know the frequency with which code is executed: some parts of the source code could actually be dead code, while others may be hotspots taking the major part of the execution time. As it stands, we cannot be sure whether the code triggering dynamic features is actually executed; some dynamic feature usage could on the other hand be executed very often. In addition, Smalltalk features a system

dictionary—a dictionary bindings names to classes—that we did not include in the study, as it would require dynamic analysis to differentiate this specific dictionary from the other dictionaries used in the code.

Our manual analysis of the 377 feature usages in our sample involves partial program comprehension, and a degree of subjective judgement. As such, it is possible that mistakes were made in the classification of the source code fragments, or that the intent of some of them was misinterpreted. We tried to avoid that by having at least 2 persons review each of the instances, and inspect closely the ones where the two judges disagreed, before taking a final decision.

Threats to external validity. Our study includes only open-source projects for obvious accessibility reasons, hence we cannot generalize the results to industrial projects.

We only consider projects that are found in the Squeaksource repository. Squeaksource is the *de facto* standard source code repository for Squeak and Pharo developers, however, we cannot be sure of how much the results generalize to Smalltalk code outside of Squeaksource, such as Smalltalk code produced by VisualWorks users.

Our corpus of analyzed projects only contains Smalltalk source code. Our hypothesis is that Smalltalk code, with the ease of use of its reflective features, constitute an upper bound on the usage of dynamic features. This assumption needs to be checked empirically by replicating this study on large ecosystems in other programming languages.

We selected the top 1,000 projects based on their size to filter out projects that might be toy or experimental projects. We believe such filtering increases the representativeness of our results, however, this might also impose a threat.

Threats to internal validity. To distinguish pure application projects from other types of projects, we classify them into sub-categories. Results show that application projects use dynamic features less often than most other project categories. However, code categorized in the crosscutting category *Tests* for instance

might use more or less dynamic features depending on the project the test code belongs to rather than on the fact that it is test code. There might be other reasons why projects categorized as applications use dynamic features less often than explained by the categorization in application and non-application code.

Threats to statistical conclusion validity. To determine whether specific kinds of projects such a system libraries or development tools use dynamic features more often than regular applications (hypothesis 2), we applied statistical tests to compare application projects to other kinds of projects (cf. Section 5.3.1). These tests are biased due to the fact that application projects are over-represented (83% of all analyzed projects belong to this category) compared to projects of the other categories. The applied t-test to a certain degree accounts for the unequal sample sizes of application and non-application projects, however, the over-representation of application projects clearly imposes a threat to conclusion validity.

To account for that in the later discussion, we defined an over-representation factor (ORF), that we use as support in the discussion on the unsafe dynamic feature usages. The ORF explicitly takes into account sample size, and allowed us to discover that for each individual dynamic feature, projects classified as Applications had less usages of unsafe dynamic features than expected. In contrast, Tests and System projects often had five times as many unsafe dynamic feature usages that one would expect. Due to the numerous tests that would have been involved (and potential type I errors associated), we did not test for statistical significance whether each individual feature was significantly more represented in each category of projects compared to applications.

5.8 Related work

There have been a number of empirical studies on the usage of programming language features by developers.

Knuth studied a wide variety of Fortran programs, informing quantitatively "what programmers really do" [Knuth, 1971]. Knuth performed static analysis on a sample of Fortran programs, and dynamic analysis on a smaller sample, recording the frequency of execution of each kind of instruction. Knuth found several possible

optimizations to compilers and suggested compiler writers to consider not only the best and the worst cases, but also the average case of how programers use language features in order to introduce optimizations.

Melton and Tempero measured the size of cycles among classes in 78 Java applications, and found that most applications featured very large cycles (sometimes in the thousands of classes) [Melton and Tempero, 2007].

Tempero *et al.* characterized the usage of inheritance in 90 Java programs, and found a higher usage of inheritance than they expected [Tempero et al., 2008]. Rysselberghe and Demeyer took evolution into account and proposed hypotheses on how the hierarchies change over time, based on observations about the evolution of two Java systems [Rysselberghe and Demeyer, 2007]. Later, Tempero analyzed a corpus of 100 Java programs in order to characterize how fields were used [Tempero, 2009]: a large number of classes had non-private fields, but less were actually accessed in practice.

Muschevici *et al.* performed an empirical study on how multiple dispatch is used in 9 applications written in 6 different languages that support it, and contrasted it with the Java corpus mentioned above [Muschevici et al., 2008].

Malayeri and Aldrich inspected 29 Java programs in order to determine if they would have benefited from structural (instead of nominal) subtyping, and found that the programs would benefit somewhat [Malayeri and Aldrich, 2009].

A large-scale study (2,080 Java applications found on Sourceforge) by Grechanik *et al.* asks 32 research questions on the usage of Java by programmers [Grechanik et al., 2010], related to the size of the applications, the number of arguments in methods, whether methods are overriden or not, etc.

Finally, Parnin *et al.* performed an empirical study of 20 Java systems [Parnin et al., 2013], with the goal of assessing how and if programmers transitioned to Java Generics. They found that adoption rates and delay to adoption varied from project to project (with some projects not adopting generics), and that usually one or two developers drove the adoption of Java generics.

Dynamic Features. In addition, several pieces of work have specifically investigated the use of dynamic features in Java, Python and Javascript.

Bodden *et al.* investigated the use of Java reflection in the case of the DaCapo benchmark suite, and found that the benchmark harness loads classes dynamically, and executes methods via reflection, causing the call graph extracted from static analysis to significantly differ from the call graph actually observed at runtime [Bodden et al., 2011]. Furthermore, the class loaders that DaCapo uses are non-standard.

Holkner and Harland investigated the dynamic behavior of 24 Python programs, by monitoring their execution [Holkner and Harland, 2009]. They found that the Python program in their corpus made a heavier usage of dynamic features during their startup phase, but that many of them also used some of these features during their entire lifetime.

Åkerblom *et al.* [Åkerblom et al., 2014] present an early study on a variety of dynamic features (*e.g.* `hasattr`, `del`, `eval` and `reload`) in Python. They dynamically analyzed the execution traces of 19 open source Python projects. They find that Python programmers use dynamic features that are hardly tractable statically. This behavior is presented in each stage of program execution (not only at startup) and in several kinds of programs.

Most directly related to our work is the study of Javascript dynamic features by Richards *et al.* [Richards et al., 2010]. They analyzed a large amount of Javascript code from popular web sites, in order to verify whether the assumptions that are made in the literature about the usage of the dynamic features of Javascript match reality. Some of the assumptions they checked were: "the use of eval is infrequent and does not affect semantics" (found to be false), or "the prototype hierarchy is invariant" (also false); most of the assumptions were found to be violated. In further work, the same authors performed a more thorough analysis of the usages of the *eval* function in Javascript [Richards et al., 2011]. Again, assumptions that *eval* is rarely used were found to be wrong. While Richards *et al.* use dynamic analysis to monitor manual interaction on 103 websites, we use static analysis on 1,000 Smalltalk projects. An innovation of our study is to consider the kinds of projects that use the features; this is particularly relevant in a live environment like Smalltalk, where the whole system is developed within itself.

5.9 Conclusions

We performed an empirical study of the usage of dynamic features in the 1,000 largest Smalltalk projects in the Squeaksource source code repository, accounting for more than 4 million lines of code.

We assessed the veracity of four high-level assumptions on the usage of dynamic features: Dynamic features are not used often (yet enough to be problematic); they are used more in certain kinds of applications than others; the most popular dynamic features are replicated in most static languages; and some of the dynamic feature usages are statically tractable.

We also analyzed in detail the usage of each of feature, producing a list of features ordered by the importance of their support for applications. Some are critical (message sending, system dictionary reading, instance creation, method compilation); others less so.

Subsequently, we performed a qualitative analysis of a representative sample of 377 usages of dynamic features, in order to understand the rationale behind each feature usage, determining whether it was possible to remove these usages, and to pinpoint limitations of the language that, if addressed, could make it possible to avoid relying on dynamic features.

We found that, if a large portion of the usages of dynamic features are genuine usages that cannot be refactored, others work around limitations of the programming language. In the absence of changes to the language, these could be replaced by more standard solutions to the same problems, albeit more complex. Finally, a significant minority of the usages are superfluous, and could be removed at a moderate cost to the programmer.

Relevance for Gradualtalk

This study provides us with useful information regarding dynamic features of Smalltalk. These results help us understand the use of `self` and reflection in Smalltalk. Hence, we make informed decisions regarding typing features: self types, effect systems and incremental type checking, and the programming idiom: symbols as methods.

On the use of self, we find that programmers use self to instantiate objects and create classes (see Section 5.4). In fact, 74% of instantiations use self, which shows the overwhelming importance of self types. Additionally, we find that classes are created at runtime. From those class creations, 70% are potentially unsafe, however most of them are located in testing code rather than in general applications. This result shows the relevance of supporting metaclasses in Gradualtalk and its interaction with self types. Finally, we conclude that self types is a primary feature for being included in Gradualtalk, and we must support the interaction of self types with metaclasses.

On the use of reflection, we conclude that reflective features of Smalltalk are rarely used (only in 1.76% of methods). Although they cannot be safely ignored, supporting them is overly-complex, and in some scenarios, those features are undecidable, *e.g.* obj perform: (user input), a reflective message with a non-deterministic input. Additionally, we find that some usages are unavoidable, because these usages are a direct mapping to the reflective feature itself. We also find that some usages can be refactored to a more statically tractable version, see Section 5.6.2, however refactoring all of them may be not be possible or practical. Although we do not directly study the idiom "symbols as methods", it uses perform: internally, and hence it suffers from the same undecidability problems. Even the introduction of a complex typing technique, such as effect systems, will not alleviate those problems because of the undecidability issues. Moreover programmers may be forced to add cumbersome annotations to get some typing benefits.

On the other hand, incremental type checking would allow the type system to handle specific class modifications, *e.g.* adding or removing a method, with significantly less effort than effect systems, *e.g.* annotations are not required. Therefore, we exclude effect systems of Gradualtalk and consider incremental type checking as a partial solution. Furthermore, the idiom "symbols as methods" will remain uncovered in Gradualtalk. In this scenario, programmers may refactor those usages or leave them untyped.

Chapter 6

On the Use of Type Predicates in Smalltalk

Object-orientation relies on polymorphism to express behavioral variants. As opposed to traditional procedural design, explicit type-based conditionals should be avoided. This message is conveyed in introductory material on object orientation, as well as in object-oriented reengineering patterns. Is this principle followed in practice? In other words, are type predicates actually used in object-oriented software, and if so, to which extent?

Answering the above questions will clarify whether complex flow-sensitive typing approaches are necessary to be included in the first version of Gradualtalk (see Section 3.4.2). In this chapter, we report on a study of the use of type predicates in Smalltalk. First, we offer an overall introduction of this study and present the research questions (Section 6.1). We then describe the experimental corpus and methodology as well as a classification of discovered predicates (Section 6.2). The following four sections (Section 6.3, 6.4, 6.5 and 6.6) report on the four research questions presented in this study, respectively. Section 6.7 discusses the threats to validity; Section 6.8 reviews related work. Finally, Section 6.9 concludes that the flow-sensitive typing feature is useful for objects. However due to a somewhat minor presence of the idiom in Smalltalk and possible interaction issues with other typing features, flow-sensitive typing will not be included in the first version of Gradualtalk.

6.1 Introduction

The object-oriented programming paradigm frees developers from manual dispatch based on explicit type predicates by relying on polymorphism. As any good object-oriented programming book tells us, messages are sent to objects and these objects react appropriately. This brings benefits in extensibility, as type case analyses do not need to be extended whenever a new kind of object is added and understands the message. Polymorphism based on dynamic method dispatch makes type predicates obsolete—at least in theory. However, recommendations, like in Effective C++[1], as well as the "Replace Conditional with Polymorphism" and "Introduce Null Object" refactoring patterns [Fowler, 1999; Nierstrasz et al., 2009] suggest that programmers do not always follow the principle, have to be reminded repeatedly, and need support to follow it more closely and make their code "more object-oriented" so as to enjoy the promised benefits.

Empirically studying the use of type predicates is relevant to both the general object-oriented programming community and the active research program of designing type systems for existing dynamic languages (*e.g.* [Guha et al., 2011; Laforge, 2012; Pearce, 2013a; Tobin-Hochstadt and Felleisen, 2008, 2010; Winther, 2011]), many of which are object oriented. Indeed, retrofitting a type system onto an existing language requires that the programming idioms embraced by developers be properly accommodated. Failing to do so compromises adoption of the retrofitted type system. Hence, the prevalence of type predicates and their common usages are important for both type system designers and the community in general.

Recently, Tobin-Hochstadt and Felleisen have made a very good case in favor of *flow-sensitive* typing to accommodate control-related programming idioms in the context of Racket, a dialect of Scheme [Tobin-Hochstadt, 2010; Tobin-Hochstadt and Felleisen, 2008]. A flow-sensitive type system such as *occurrence typing* is able to account for the type information gathered in the use of type predicates in conditionals. For instance, consider the following Scheme definition:

[1] *Anytime you find yourself writing code of the form "if the object is of type T1, then do something, but if it's of type T2, then do something else," slap yourself [Meyers, 2005]—Scott Meyers.*

```
; x is a number or a string
(define (f x)
  (if (number? x)
      (add1 x)
      (string-length x)))
```

The function f accepts either a number or a string; if given a number, it adds 1 to it; if given a string, it returns its length. Knowing if the argument is a number is determined by the function number? (of type $Any \to Boolean$). In order to type this method, the type system must be able to understand that the application of add1 (of type $Number \to Number$) is valid, because at this point x is necessarily a number; similarly for the application of string-length.

Occurrence typing was later extended with *logical types* in order to account for the logical combination of predicates in conditionals [Tobin-Hochstadt and Felleisen, 2010], *e.g.* (or (number? x) (string? x)). The resulting type system is expressive but complex. In addition, scaling to a language with objects and mutable state requires even more complex flow analysis [Guha et al., 2011; Pearce, 2013a; Winther, 2011].

Guha *et al.* [Guha et al., 2011] propose flow typing, a type system for JavaScript that relies on control flow analysis to properly type variables in control flow statements, for instance:

```
var state = undefined;
...
function updateState() {
  if (typeof state === "undefined") {
    state = 0;
  }
  return state + 1;
}
```

The above code is the classic example of a lazy initializer. The function updateState checks if the variable state is undefined and if so then state is initialized with 0, otherwise, state is a number and it can be (safely) incremented.

Flow-sensitive typing approaches are not only beneficial for retrofitted type systems, but also for existing type systems. This is the case of *Guarded Type Promotion* [Winther, 2011], a type system extension for Java that tracks instanceof occurrences in control flow statements to remove unnecessary casts. For instance:

```
if (obj instanceof Foo) {
  ((Foo) obj).doFooStuff();
}
```

In Java, casting the variable obj to Foo is necessary to properly call method doFooStuff. However, with Guarded Type Promotion, the variable obj can be safely considered an instance of Foo, and hence all Foo's methods can be called. An improved version of the code is:

```
if (obj instanceof Foo) {
  obj.doFooStuff();
}
```

The question arises whether or not these techniques are practically useful in an object-oriented setting, where type predicates are supposedly avoided. Interestingly, most if not all object-oriented languages provide operators to do runtime type checks, like Java's instanceof. Their use is however strongly discouraged, with the only exception being for implementing binary equality methods [Bloch, 2008]. Binary methods are indeed well-known to be hard to properly implement in an object-oriented language [Cook, 2009]. But if flow-sensitive typing is only helpful for equality methods, one could reasonably argue that its complexity cost trumps its static typing benefits.

Contributions. In order to shed light on these questions, we perform an empirical study of the use of type predicates in the dynamic object-oriented language Smalltalk. Smalltalk is a pure object-oriented language: everything is an object, even classes, and control structures are the results of sending messages[1]. Furthermore, the Smalltalk main libraries have been designed with a strong object-oriented focus. Because of this, one might expect that Smalltalk programmers tend to produce code embracing object-orientation. We analyze 1,000 open source Smalltalk projects, featuring more than 4 million lines of code. Our study reveals if, and how, type predicates are used in practice. We answer the following research questions:

RQ1: How prevalent is the use of type predicates to do explicit dispatch? This question directly addresses the main question of this chapter,

[1]Strictly speaking, basic control flow structures in Smalltalk are handled directly by the VM for optimization purposes.

with respect to how much the principle of relying upon polymorphism instead of type predicates is followed in practice. This informs type system designers on the usefulness of flow-sensitive typing for object-oriented programs.

RQ2: What are the different forms of type predicates used? Are some categories largely predominant? Solving specific problems is often easier than solving general ones. Answering these questions allows us to understand if ad-hoc type systems handling specific cases (*e.g.* non-null types [Fähndrich and Leino, 2003]) would be "good enough".

RQ3: How prevalent is the use of logical combinations of type predicates? Logical types [Tobin-Hochstadt and Felleisen, 2010] allows type systems to properly handle type predicates composition using logical combinators, as described above. This question sheds light on whether this technique would be of significant value in object-oriented software.

RQ4: Are identified (type) predicates constant? Object-oriented languages usually support mutable state, which makes occurrence typing unsound if a (type) predicate is not constant. Evaluating the prevalence of this issue informs if more complex techniques like flow typing [Guha et al., 2011; Pearce, 2013a; Winther, 2011] or typestate checking [DeLine and Fähndrich, 2004; Strom and Yemini, 1986] are necessary.

6.2 Experimental setup

This section describes the corpus of projects we are analyzing, the methodology applied to find predicates, and a classification of the discovered predicates.

6.2.1 Corpus

We analyze a body of 1,850 projects, which we used previously in Chapter 5 (see Section 5.2.1), where we studied the use of reflective features. To exclude small or toy projects we ordered all projects in the entire corpus by size (LOC) and selected the 1,000 largest ones. Our corpus is a snapshot of the Squeaksource Smalltalk repository taken in early 2010. The corpus includes a total of 4,445,415 lines of

code distributed between 47,720 classes and 652,990 methods. The largest project is Morphic, with 124,729 lines of code.

In order to analyze the projects, we use the Ecco model [Lungu et al., 2010], a lightweight representation of software systems and their versions in an ecosystem, allowing for the effective analysis of interdependent systems. We extend our previous framework (see Section 5.2.1) to statically trace the declarations and usages of type predicates in the software ecosystem[1].

6.2.2 Finding predicates and their usages

What is a type predicate? We are interested in tracking usages of Smalltalk's equivalent of Java's instanceof, named isKindOf:, or some variants thereof. Also as in Racket, we are interested in functions like string?, which in Smalltalk would be defined as polymorphic methods (*e.g.* isString). When there are multiple ways to express the same check, we do our best to detect all forms. We distinguish four categories of predicates, all described below.

Nominal. Smalltalk natively provides a number of ways to check the type of an object. This category corresponds to *nominal* type checks, *i.e.* related to the actual class of an object. The equivalent of Java's instanceof operator is called isKindOf:. A strict version, isMemberOf:, checks if an object is a direct instance of the given class (without considering subclasses). For example:

```
'a text' isKindOf: Object      "returns true"
'a text' isMemberOf: Object   "returns false"
```

Additionally, we also count type checks performed through explicit class comparison (reference equality ==, user-defined equality =, and non-equality ~=). Eg:

```
'a text' class == String    "returns true"
```

Structural. Like many other dynamically-typed object-oriented languages, Smalltalk also supports *structural* type checks using respondsTo: or canUnderstand:. These checks are used to determine if an object understands a given message, regardless of its implementing class. For instance:

[1]This extension is available at http://ss3.gemstone.com/ss/TOC/

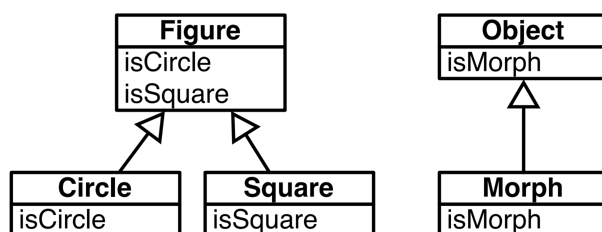

Figure 6.1: Examples of polymorphic type predicates.

```
true respondsTo: #not            "returns true"
Boolean canUnderstand: #+        "returns false"
```

Polymorphic. Polymorphic type predicates are methods that play the role of type discriminators, just like string? in Racket. Figure 6.1 shows two class hierarchies with type predicates. In class Figure, both isCircle and isSquare return false; they are overridden in their respective subclass to return true. The case of Morph is similar, but showcases the use of class extensions (*aka.* open classes) in Smalltalk. The isMorph method is added to Object and is overridden in Morph. In Smalltalk, the Object class is routinely extended with such external methods (57% of the packages contained in the Pharo distribution extend a class defined in another package and 9% of Pharo packages extend Object).

This category of predicates is therefore user-extensible, and we need a heuristic to detect them. Following the Smalltalk naming conventions, a type predicate is a selector (method name in Smalltalk jargon) that follows the pattern isXxxx—the prefix is the verb is, followed by any camel-case suffix. Often the suffix is the name of a class (or part of it), but it can be any other string. We only consider methods that do not have any arguments. The body of a type predicate method should return a literal Boolean in all of its implementations. Usually the returned Boolean is false in a superclass and true in a subclass.

The above heuristic is admittedly very conservative. However, if we include all isXxxx methods, some of them correspond to *state* rather than *type* abstractions; and the boundary can be hard to draw. For instance, isEmpty can be implemented as a state predicate or as a type predicate depending on the chosen design.

Nil predicate. Nullity checking is supposedly a prevalent activity in object-oriented languages, which has triggered a number of efforts to design languages with non-null types [Fähndrich and Leino, 2003]. Smalltalk provides the nil value as a unique instance of the singleton class UndefinedObject. The nil predicate isNil is in fact implemented as a polymorphic predicate. We group all nil-related predicates provided by the language (*e.g.* notNil), as well as related control flow expressions, such as ifNil:, ifNotNil:, etc. Additionally, we also include explicit nil equality checks in this category, *e.g.* obj == nil.

6.3 Prevalence of type predicates

To address the question of the prevalence of type predicates and their usage, we start by reporting on the results of our predicate detection algorithm, and then classify predicate usages in order to refine our analysis.

6.3.1 Basic statistics in Squeaksource

Our predicate detection algorithm identified 1,524 different polymorphic predicates. This represents 0.6% of all selectors (method names) in the corpus. As mentioned above, we detect isXxxx methods regardless of whether Xxxx actually corresponds to a class name (or part of one). We find that almost one out of five predicates do not match a class name (19.1% – 245). These predicates are important because they represent (type) abstractions that crosscut the class hierarchy; in Java, one would expect these to be represented as interfaces. Examples include isShape, isDisplayable, and isAnnotation.

All predicates (nominal, structural, polymorphic and nil) are used 107,897 times in our corpus, spread out in 971 out of the 1,000 projects we considered. Only 631 usages (0.6%) occur inside equality methods (=, ~=, closeTo:, and literalEqual:), suggesting that the recommendation of using type checks only in equality methods [Bloch, 2008] is rarely followed in practice. The issue is hence quite widespread and awareness of it needs to be raised among practitioners. Additionally, this result suggests that flow-sensitive typing would be helpful beyond equality methods, if these usages are indeed in a control flow or similar statement, *e.g.* an assertion.

These usages described above could occur in object oriented software due to several reasons. Here, we present a non exhaustive list beyond design faults.

- *Legitimate usages.* Some usages are legitimate and cannot be avoided mainly because of limits of the language, *e.g.* checking whether or not a variable has been initialized.
- *Convenience.* In some scenarios, especially in small operations, it may be simpler to use type based dispatch rather than creating polymorphic methods.
- *Evolution.* Some authors [Krishnamurthi et al., 1998; Oliveira, 2009; Robbes et al., 2012; Torgersen, 2004; Zenger and Odersky, 2005] report that object-oriented software not only evolve by adding new classes, but also by adding new methods. Some type predicate usages could indicate an anticipation of this evolution.

We actually do not include all 107,897 usages in our study, because some usages do not impact the flow of the program in a way directly observable to our static analysis. The next section introduces the classification of predicate usages on which our refinement is based.

6.3.2 Usage categories

We classify usage contexts of type predicates as follows:

- **Dispatch.** The predicate is clearly used to drive control flow in ifTrue:ifFalse, whileTrue, doWhileTrue, etc. Eg:

  ```
  figure isCircle ifTrue: [figure radius] ifFalse: [figure width]
  ```

  This correponds to the classical examples where flow-sensitive typing is beneficial.

- **Collections.** The predicate is used to filter or test elements inside a collection, with select:, reject:, detect:, allSatisfy:, etc. For instance:

Usage context	Usages	(%)	Selected	(%)
Dispatch	86,561	(80.2%)	79,837	(92.2%)
Collections	3,179	(2.9%)	3,179	(100%)
Assertions	10,964	(10.2%)	10,220	(93.2%)
Forward	4,994	(4.6%)	0	(0%)
Others	2,199	(2%)	0	(0%)
Total	107,897	(100%)	93,236	(86.4%)

Table 6.1: Usage categories of type predicates with their refinements.

figures select: #isCircle returns all circles in the figures collection. A flow-sensitive type system can then keep track of this information, validating invocations of circle-only methods on elements of the returned collection.

- **Assertions.** The predicate is used in an assertion context, such as assert or deny. Eg: figure isCircle assert. This expression is similar to a conditional where the false branch raises an error. The next statement after the assertion can in fact use the fact that figure is a circle.
- **Forward.** The predicate is used to define another predicate, *e.g.* Figure>>isOval ↑ self isCircle
- **Others.** The catch-all category for usages that do not fit in any of the previous ones.

Table 6.1 shows the number of raw usages and the percentages of usages (second column) categorized by usage context (first column). Unsurprisingly, simple conditional dispatch is the most common usage idiom, with 80.2% overall usages, and a presence in 94.8% of the projects. Then comes Assertions at 10.2%, showing that type predicates are often used in testing contexts, or in pre/postconditions. The three other categories are relatively scarce.

A Note on Collections. The Collections category represents only 2.9% of usages. This low value was actually contrary to our expectations. The empirical study on the use of collections (see Section 4.5), especially the dynamic analysis, can help us understand why there is a low number of uses. In this study, we find that 94.6% of collections are homogenous, which means their elements are instances of the same class. While the results are not representative of all the projects (only the default

image and Seaside), the homogeneity of collections appears to be a good reason why type predicates are seldom used when operating over collections.

6.3.3 Refinement

For the remainder of this study, we keep only a selected group of predicates from the Dispatch, Collections, and Assertions categories, as we want to focus only on those predicates whose flow-sensitive typing approach can benefit programmers (third column in Table 6.1).

In the Dispatch and Assertions categories, we filter out those predicates where flow-sensitive typing may not be relevant, as described below. The static analyzer tracks locally if there is at least one usage of the receiver in the statements or expressions following the predicate check. This heuristic is an approximation, but more powerful analysis is very expensive to perform in Smalltalk, see a more detailed discussion in Section 6.7. Some filtered out examples (extracted from the corpus) follow:

```
moduleExtension
 ↑ self isCPP ifTrue: ['.cpp'] ifFalse: ['.c']


initialize
 "Initialize the OpenGL context, required by AmanithVG"
 | renderer |
 self assert: VG isNil.
 renderer := self getAPIRenderer.
 accelerated := renderer beginsWith: 'AmanithVG GLE'.
 ...
```

In the first method `moduleExtension`, the type information provided by the predicate `isCPP` is not used in any of the branches. Similarly in the method `initialize`: The information `VG isNil` is not directly exploited in the remaining statements. However, any called method may use that information, but tracking that in a static analysis is hard to achieve.

For usages in the Collections category we keep all usages, because tracking non-relevant usages in a highly-dynamic language like Smalltalk is complex to achieve. Furthermore, even a high fraction of non-relevant usages would not significantly affect our study due to the low percentage of usages in this category.

The Forward and Others categories are completely excluded because it is not clear how these usages impact the control flow of the program. Usages in the Forward category correspond to the use of predicates to implement another type predicate; the type predicate they are a part of is still referenced and counted as a normal type predicate. We can see that there is a small, but significant effort devoted to reusing existing predicates in order to define others.

The last excluded category, Others, contains all usages of type predicates that do not fit in the top four categories. Its small size, 2%, tells us that our classification is quite exhaustive: there are no obvious categories that we are missing. The results we report in this chapter will at worst be a slight under-estimation of the actual usage of predicates. Looking at the common idioms in this catch-all category, we found that more than half of them consist in storing the value of a predicate in a variable for later use, or were passed as arguments to other methods. These cases would require a significantly more advanced static analysis to precisely track them, hence our preference for under-estimation. Other cases are more arcane, *e.g.* reflective predicate invocation, or are clearly not predicates, *e.g.* a predicate is called but the returned value is not used. The rarity of the latter case tells us that our heuristic of considering isXxxx methods as predicate is correct, since a very large majority of the predicates *are* used as type predicates.

Taken together, the three usage categories we select comprise more than 86% of the predicate usages we encountered. From this, we can conclude that predicates are indeed used in order to impact the control flow in a direct way that would be easily exploitable by a flow-sensitive type system. Alternatively, refactoring the source code to replace conditionals with polymorphism has the potential to reduce complexity in a large number of cases. But assertions cannot be refactored and refactoring is not a solution for a retrofitted type system.

6.3.4 Prevalence of predicate usages

After refinement, we are left with 93,236 usages of type predicates that directly affect the control flow of programs. We now assess whether this number means that type predicates are prevalently used or not. We evaluate the presence of type predicate usages at different levels of granularity: projects, classes, methods and

lines of code (Figure 6.2). Indeed, recent work by Posnett *et al.* has shown that observations that hold at one level do not necessarily hold at others, leading to the risk of committing an *ecological fallacy* [Posnett et al., 2011].

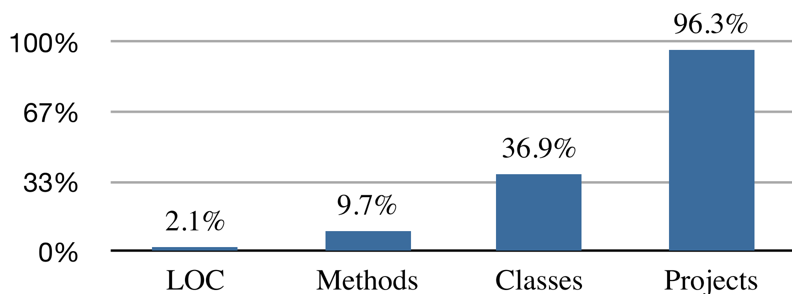

Figure 6.2: Presence of type predicates in LOC, methods, classes and projects.

First, how does the number of usages translate in terms of actual prevalence in Smalltalk projects? We find that 96.3% of projects use type predicates, *i.e. not* using type predicates is the exception rather than the rule.

At the level of classes, we find that slightly more than a third—36.9%—of the classes use type predicates as part of their implementation. This figure supports the claim that programmers use type predicates quite commonly.

At a finer-grained level, we find that 9.7% of the methods are using type predicates. Again, this confirms the previous finding, as this is certainly a large minority of all the methods. Clearly, flow-sensitive typing has the potential to provide more accurate type information in the control flow of one out of ten methods.

But perhaps the most telling figure is the finest-grained one, which is the density of type predicate usages per lines of code, telling us how many lines of code we might expect to read before encountering a usage of a type predicate. Considering that we have referenced 93,236 usages of type predicates in the 4,445,415 lines of code in our corpus, we find a density of 0.021 predicates per line of code, or 2.1%. Considering a homogenous distribution, one might expect to read around 50 lines of code to encounter a type predicate usage. This further highlights that usages of type predicates are a common sight in object-oriented source code, and that better supporting them would have a practical impact on the daily work of programmers. The advice of avoiding these type-checks is not followed in practice.

6.3.5 Summary

We find many conditional dispatches based on type predicates in source code. After filtering indirect and irrelevant usages, we find that almost 10% of all methods do explicit type-based dispatch, and that the density of type predicates per lines of code is 2.1%. These findings highlight the opportunities for flow-sensitive typing mechanisms such as occurrence types in object-oriented programs.

6.4 Prevalence of categories of type predicates

Beyond the overall prevalence of type predicates, we are interested in the prevalence of specific *categories* of predicates, as described in Section 6.2.2. Are certain categories of type predicates more commonly used than others? If that is the case, this allows us to make informed decisions: varying cost and challenges in the implementation of a type system that supports it fully. Alternatively, it may indicate that the type predicates issue is more prevalent in certain scenarios.

6.4.1 Predicate categories

Kinds	**Usages**	**% Usages**	**% LOC**	**% Methods**	**% Classes**	**% Projects**	**% Logical**
Nominal	14,518	15.6	0.3	1.4	8.4	64.5	19.8
Structural	1,397	1.5	0.03	0.2	1.5	26.5	11.7
Polymorphic	6,446	6.9	0.15	0.7	4.5	41.4	28.5
Nil	70,875	76.0	1.6	8.0	32.2	95.0	11.2
All	93,236	100.0	2.1	9.7	36.9	96.3	13.8

Table 6.2: Usages distributions for coarse and fine-grained predicate categories.

Table 6.2 shows the distribution of each predicate category (nominal, structural, polymorphic and nil) by usages among all projects. We clearly see the categories of predicates are not equally distributed. The Nil predicate takes the largest share at 76% (70,875) of all usages, nominal type predicates follow with 15.6% (14,518), polymorphic and structural type predicates only amount to 6.9% (6,446) and 1.5% (1,397) of the total usages respectively.

Distribution at different levels of granularity. The analysis above is reflected in the distribution in terms of frequencies of presence in projects, classes, methods,

and LOC, which is shown in Table 6.2 as well. Most of the usages at all levels are Nil predicate usages. As we observed above, the proportion of projects that use a given type predicate is much higher than the proportion of classes, methods, or LOCs. Aggregating at the project level does not give a complete picture; it only tells us that a vast majority of projects use nil-related predicates, but not how much they are used. Likewise, structural predicates are used by more than a quarter of the projects, but are used very sparsely at the class, method or LOC levels. At the method level, nominal and polymorphic type predicates are used in 1.5% and 0.7%, respectively, making their usages more frequent than structural type predicates, but still fairly localized in the corpus.

6.4.2 Usage contexts and predicate categories

Kinds	**Dispatch (D)**	**Assertion (A)**	**Collections (C)**	**% (D)**	**% (A)**	**% (C)**	**% (D)**	**% (A)**	**% (C)**
Nominal	9,370	3,768	1,380	11.7%	36.9%	43.4%	64.5%	26%	9.5%
Structural	1,271	41	85	1.6%	0.4%	2.7%	91%	2.9%	6.1%
Polymorphic	4,624	972	850	5.8%	9.5%	26.7%	71.1%	15.1%	13.2%
Nil	64,572	5,439	864	80.9%	53.2%	27.2%	91.1%	7.7%	1.2%

Table 6.3: Usage contexts and predicate categories: The first group of three columns shows the number of usages by context and category. The second group shows the distribution of usage contexts by predicate categories (columns sum 100%). The last group shows the distribution of predicate categories by usage contexts (rows sum 100%).

Table 6.3 shows the usages by context and predicate category (first group of three columns) with their respective distributions (second and third group). The distribution of predicate categories by usage contexts (second group) shows:

- In the dispatch context, nil predicates takes the largest share (80.9%), nominal comes second (11.7%), and polymorphic and structural at the end (5.8% and 1.6%, respectively). This distribution has a big influence on the overall distribution, because dispatch usages account for more than 80% of all usages.

- In the assertion context; nil predicates account for a bit more than half of the usages (53.2%); nominal and polymorphic predicates take almost the other

half (46.4%) with 36.9% and 9.5%, respectively; finally structural predicates are rarely used in assertions (0.4%).

- The usages in the collection context are the most interesting. Nominal predicates take the largest share with 43.4%. Nil and polymorphic predicates are almost equally distributed with 27.2% and 26.7%, respectively, and structural predicates are last with only 2.7%.

The distribution of usage contexts by predicate categories (third group) shows almost a similar distribution in each predicate category. Dispatch usages are the most prevalent ranging from 64.5% in nominal predicates to 91.1% in nil predicates. Assertion usages come second in nominal (26%), polymorphic (15.1%) and nil (7.7%) predicates; the only exception is structural predicates, where assertion usages appear last with 2.9%. Collection usages rank last in all categories but structural (6.1%). Particularly, nil predicates are rarely used in a collection context (1.2%).

6.4.3 Nil predicate

Since nil-related predicates are so prevalent, we investigate them further. In Table 6.2, we see that the Nil category consists of more than three quarters of all predicate usages (76% or 70,875 usages). If we look at the distribution of usages of nil predicates, we note that 8% of all methods include a usage of a nil predicate (a density per lines of code of 1.6%). Additionally, more than 90% of nil usages are in a dispatch context (see Table 6.3), which makes it even more easy to apply a non-null type technique.

Tony Hoare's self-admitted "billion-dollar mistake"[1] is hence alive and well in Smalltalk code. On the upside, this presents opportunities for enhancement. One can clearly see how a type system with non-null types would be beneficial in a slightly more than three quarters (76%) of the cases we found in our corpus.

[1] http://tinyurl.com/hoare-mistake

6.4.4 Polymorphic predicates

Almost 7% of all predicates are polymorphic predicates. These type predicates are roughly half as prevalent as the nominal category. Combining these usages with the usages of nominal type predicates, nominal type predicates can be seen as polymorphic type predicates waiting to be, we end up with 20,964 usages, or 22.5% of all usages; a bit more than a fifth. This indicates a potential usefulness of a type system able to handle arbitrary type predicates. A good example is Typed Racket [Tobin-Hochstadt and Felleisen, 2008] with occurrence typing.

Additionally, from Table 6.3, we can see that nominal and polymorphic predicates account for 70.1% of all usages in a collection context. This strengthens the necessity of flow-sensitive typing to support collection usages and not only direct control flow usages.

6.4.5 Summary

Another way to look at the results is "what is the best bang for the buck" in terms of implementation effort. Here, we see that a type system solely dedicated to handle nil-related predicates would have a very broad applicability, as this category totals more than 70,000 type predicates usages, which accounts for 76% of all usages. The results also tell us that a vast majority of nil checks (90.1%) occur in a direct control flow statement. As for full-blown flow-sensitive typing supporting polymorphic, nominal and structural predicates, the results suggest that it is still worthwhile, in order to cover the last quarter of all usages. Such a type system would also cover the nil case.

6.5 Prevalence of logical combinations

To assess the practical value of logical types [Tobin-Hochstadt and Felleisen, 2010] in an object-oriented context, we now study to what extent predicates are used in complex expressions combined with logical combinators.

Logical combinators are the boolean operators, which in Smalltalk includes `and:`, `or:` and a variety of sibling selectors (*e.g.* `&`, `and:and:`). Predicates can be composed with others by using such logical combinators to produce more refined

or detailed predicates. As such, usage of logical combinations of type predicates cuts across predicate categories. An example of using logical combinations of type predicates is:

```
expr isMessage and: [expr receiver isVariable]
(prefix isKindOf: String) & (suffix isKindOf: String)
```

6.5.1 Overall prevalence of logical combinations

We found that a significant portion of all usages of type predicates are included in such logical combinations. As the last column of Table 6.2 shows, out of the 93,236 occurrences of type predicates we found in the corpus, 13.8% are part of logical expressions. A sizable minority of all type predicate usages is part of a more complex logical predicate expression, suggesting that a flow-sensitive type system should indeed account for such combinations. However such type systems are usually complex to use, because of the additional annotations and the extra effort required to understand them. Finally, the proportion of logical type predicates varies with the type predicate used.

6.5.2 Prevalence in nil predicates

In particular, nil predicates are much less present (11.2%) in composed expressions than any other type predicate groups. More than in other categories, it is very common to discriminate for the null value only. Hence, the emphasis on non-null types would have an important impact for a comparatively low effort: many type predicates testing for the null value are executed in isolation and are not embedded in a logical combinator (88.8%).

6.5.3 Nominal and polymorphic predicates

On the other hand, considerably more nominal and polymorphic type predicates than nil predicates are to be included in conditional expressions (with 19.8 and 28.5%). A possible reason for such a higher proportion is that a *simple* conditional dispatch based on the type of the object is more likely to be refactored to a

polymorphic method, since the infrastructure to host the polymorphic method—the hierarchy of classes where it has to be implemented—is already present. As a result, a higher proportion of *complex* logical type predicates are present. Still, the fact that more than 70% of the cases are simple conditionals means that polymorphism is not used as much as it could be. Given that these categories of predicates account for more than a fifth of all usages, the additional complexity occasioned by the largest proportion of logical combinations makes the task of implementing a type system handling arbitrary type predicates more dependent on the additional inclusion of logical types.

6.5.4 Structural predicates

Considering structural type predicates, we see that the proportion of logical type predicates is lower, at 11.7%. One possible reason for this is that programmers mostly use these predicates to check if an object understands a specific message in order to immediately send it, and usually not to perform more complex operations.

6.5.5 Summary

Logical combinations of type predicates are indeed prevalent in object-oriented source code. However, they are more prevalent in polymorphic and nominal predicates, than in nil type predicates. This makes the decision whether or not to support logical types somewhat dependent on the initial type system considered. Our results concur with Tobin-Hochstadt and Felleisen [Tobin-Hochstadt and Felleisen, 2010] in that a full-blown flow-sensitive type system should account for logical combinations of predicates.

6.6 Prevalence of constant predicates

Object-oriented languages usually support mutable state. If a predicate is based on a mutable state, then it may not be constant. As a consequence, occurrence typing as originally formulated [Tobin-Hochstadt and Felleisen, 2008, 2010] is unsound if predicates vary over time. Advanced approaches like flow typing [Guha et al.,

2011; Pearce, 2013a; Winther, 2011] and typestates [DeLine and Fähndrich, 2004; Strom and Yemini, 1986] can soundly account for type predicates with mutable state but at the cost of increased complexity. It is therefore important to evaluate how problematic this issue is in practice.

6.6.1 Classification of predicates

In order to assess the prevalence of the issue related to mutable state, we look at how polymorphic predicates are implemented.[1] We first focus on the static analysis of all the predicate implementations of the corpus:

- We consider a predicate implementation to be *statically constant* if its body returns a literal boolean (true or false).[2] Eg:

  ```
  Circle>>isCircle
    ↑ true
  ```

  These predicates (1,524) are the polymorphic predicates that we analyzed in the previous sections.

- Otherwise it is considered to be *potentially variable*. For instance, the following predicate implementation relies (indirectly) on an instance variable (which is, in fact, mutable): Eg:

  ```
  File>>isOpen
    ↑ fileDescriptor notNil
  ```

  These predicates (2,989) are polymorphic predicates that we initially discarded because they have at least one state-based implementation, as in the example above.

In the remainder of this analysis, we use the terms *constant* and *variable* in the meaning described above. This classification is a safe under-estimation of the

[1]Strictly speaking, in a very dynamic language like Smalltalk, even a nominal check with isKindOf: cannot be relied upon soundly, because the class of an object can be changed dynamically. However, our previous study of the use of such reflective features in Smalltalk shows that these cases are marginal (see Section 5.4).

[2]We considered the case of logical combinations of constant predicates; however we found only one of these cases in the corpus.

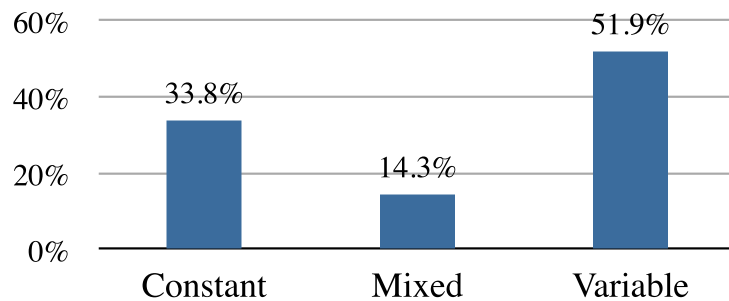

Figure 6.3: Predicates distribution based on constancy.

number of constant predicate implementations, as mentioned in Section 6.2.2. This means that we may qualify certain implementations as variable even though they are in fact constant. In Section 6.6.5 we report on a dynamic analysis of a subset of predicates that refines the classification.

A given polymorphic predicate can be implemented in several classes,[1] sometimes in a constant manner, and sometimes not. Because we are interested in the constancy of predicates in general (not of a given specific implementation), we perform the following classification:

- **A constant predicate** is a predicate for which *all* implementations are constant.
- **A variable predicate** is a predicate for which *all* implementations are variable.
- **A mixed predicated** is a predicate for which some implementations are constant, and some are variable.

For the sake of occurrence typing, only reasoning on the use of constant predicates is sound.

[1] For 4,513 predicate names, we count 8,573 implementations, meaning that a predicate is implemented 1.9 times on average.

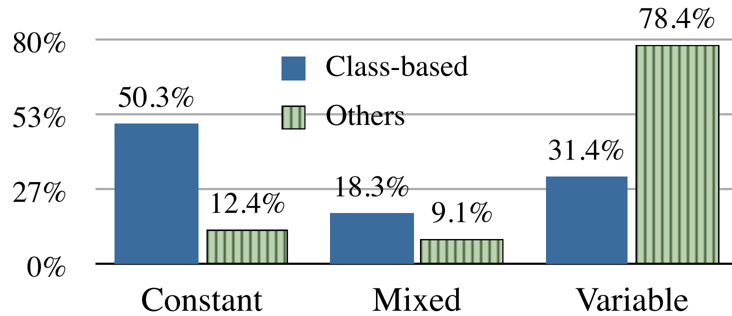

Figure 6.4: Refined constancy distribution, depending on predicate name.

6.6.2 Prevalence of constant predicates

Figure 6.3 shows the distribution of predicates based on constancy. Constant predicates account for a third (33.8%—1,524 predicates). Variable predicates account for more than half of all predicates (51.9%—2,342 predicates), and few predicates are mixed (14.3%—647 predicates). On average, these mixed predicates are implemented in 12.8 methods of which half are constant (52.8%), *i.e.* returning a literal boolean, and half variable (47.%) implementations.

These numbers suggest that the soundness issue of occurrence typing in presence of mutable state is a practical problem. Even though the fact that a third of the predicates are constant is an under-estimation, the results suggest that a good majority of the predicates are possibly variable.

6.6.3 Relevance of predicate names

We observed that predicates that *are* based on the name (or a part of it) of an existing class in the system are significantly more likely to be constant than the predicates that do not include a class name in their selector name. Figure 6.4 shows that half of the class-based type predicates are in fact constant, while this is the case for only 12.4% of the ones that are not class-based. As such, we can see that the name of a predicate could be an indicator of the constancy of its implementations.

6.6.4 Relationship between constancy and usage

Type predicates issues can be exacerbated if variable predicates are used more than constant predicates. Because of this, we analyzed whether there is a correlation between the level of constancy of the predicates (defined by the ratio of constant vs. variable implementations) and their usages. A Pearson correlation between the number of usages and the level of constancy was however found to be both extremely low (0.05) and non-significant ($p > 0.73$), confirming the (somewhat expected) absence of a relationship between the constancy of a predicate and its use.

6.6.5 Dynamic analysis of predicates

The static analysis results are broad in that they cover the whole corpus, but are arguably overly conservative. The question that naturally arises is: *how many of these mixed and variable predicates are effectively constant?*

To answer this question, we use a runtime analyzer to understand how predicates actually behave during execution. One possible way to measure predicate usages is to manually execute applications while analyzing how predicates behave at runtime. But manually and meaningfully executing 1,000 projects is not practical. Therefore, we analyze the execution of unit tests associated with each project. Using unit tests as scenarios for dynamic analysis has been reported also in other tools [Roberts et al., 1997; Thies and Bodden, 2012]. However, we are aware that unit test scenarios may be biased, and consequently, our dynamic analysis could be just a lower bound approximation.

From the 1,000 projects of the corpus, 562 offer a set of unit tests that can be used to dynamically analyze predicates. Loading and running these projects is a difficult task to automate. Each project is likely to depend on some other projects to form a runnable system. We solve this problem by extracting the dependencies from the source code, following previous work [Lungu et al., 2010]. We then run the unit tests. Of the 562 projects, only 164 are loadable and include an executable test suite. The remaining 398 projects could either not be properly loaded or executed. There are several reasons for this: not all the dependencies

can be satisfied; some projects are unstable; or the version of the base system expected by each project is not known and cannot be inferred easily.

We successfully analyzed the execution of 6,369 tests, in which 240 polymorphic predicate implementations were executed (from a total of 1,422 implementations). These 240 predicate implementations are grouped into 194 unique predicate names per project. We filter out predicate names that do not have *all* their implementations executed to meaningfully categorize the predicates. This leaves us with 164 predicate implementations of 137 unique predicate names. We found 5 predicates, discarded in the following analysis, that returned non booleans during their execution (this confirms that our false positive rate is low).

We classify each predicate into one of four categories:

- *Statically constant* predicates are, as in the static analysis, predicates whose body returns a boolean literal.
- *Constant* predicates are predicates that were classified as potentially variable, but that always return the same result across all executions.
- *Constant per object* are predicates that always return the same result for a given receiver object.
- *Variable* predicates return different results for the same receiver object.

Figure 6.5 presents the results of analyzing the execution of the 132 predicates. While only 33 (25%) predicates are statically constant, 73.4% appear constant at runtime. More precisely, 39 (29.5%) are constant regardless of the receiver, and 25 (18.9%) are constant per receiver. The remaining 35 (26.5%) are in fact variable.

This means that barely more than a quarter of the predicates in our sample are truly problematic[1], and would make occurrence typing unsound. This is because the object may mutate after the predicate check invalidating the assumptions of occurrence typing. For these cases, a more powerful typing approach that handles mutability, such as typestates, would be required.

[1]The dynamic analysis is based on unit tests that may not be representative or may be biased. Hence, the total number of constant and variable predicates may be just an approximation, see Section 6.7 for a wider discussion.

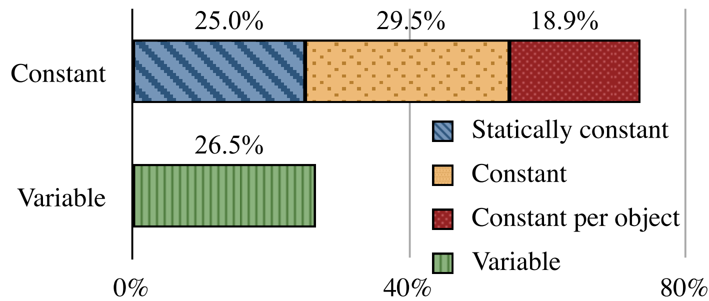

Figure 6.5: Refining constant and variable predicates with the dynamic analysis.

6.6.6 Summary

The static analysis shows that a large majority of predicates (51.9%) is potentially variable, which would require complex typing techniques such as typestates to provide a benefit for programmers. Only a third of the predicates are statically constant. However, our dynamic analysis of a sample of the predicates reveals that many of the potentially variable predicates actually behave like constant predicates. Almost 3/4 of the predicates are found to be constant for the lifetime of the objects, reducing the truly variable cases to 26.5%. Of course, our dynamic analysis may be incomplete, and our sample may not be representative. However, the fact that we found a *lower* number of statically constant predicates in our sample of dynamically analyzed projects makes us think that this estimate borders on the conservative.

6.7 Threats to validity

Construct Validity. For the main part of this study, we only use static analysis as it is impractical to perform dynamic analysis on all 1,000 projects due to reasons given in Section 6.6.5. We therefore cannot be sure whether the declared usages of predicates are actually exercised. Some identified usages may actually never be used during the execution. However, we expect the percentage of such "dead usages" to be low and to not significantly bias the results, which is also confirmed by the dynamic analysis we performed in 164 out of the 1,000 projects.

The algorithm to identify predicates might not completely cover all predicates. In particular the list of language-defined predicates we consider (isKindOf:, canUnderstand:, or isNil) might not be exhaustive. Similarly, considering only selectors following the pattern isXxxx may ignore predicates following a different naming schema.

On the one hand, we thoroughly studied the Smalltalk language to not miss any language-defined predicate in our list, and as such are confident our predicate list is exhaustive.

On the other hand, we are also aware of other method prefixes associated with predicates (*i.e.* canXxxx, shouldXxxx, hasXxxx, doesXxxx). These prefixes are not always reliable markers of type predicates, rather denoting state-based abstractions. The is prefix carries a connotation of a type—an *is-a* relation— and hence generally tells us something about what the object *is*. Prefixes such as has, can, should do not carry that connotation, having more to do with properties or capabilities that the object has.

Thus, we chose to under-estimate the prevalence of type predicates, instead of over-estimating it by including these additional prefixes. We however investigated how much this choice impacts our results. We found 1,535 defined method names matching the prefixes above. However, only 117 (7.6%) return a literal boolean in all of their implementations, totaling 1,129 usages of these predicates, in all usage contexts (including the ones we discard). In contrast, the corpus contains thirteen times more selectors and almost seven times more usages of polymorphic isXxxx predicates.

Carrying out the same analysis we performed for RQ4, we found that only 7.6% of the implementations of potential predicates with alternative prefixes were definitely constant (*i.e.* returning a literal boolean—4.5 times less than their isXxxx counterparts), while 84.75% were never literally returning a boolean (compared with 51.9% of their isXxxx counterparts). The fact that there were 7 times fewer usages, and the fact that a large majority of them seem to be state-based, makes us confident that our under-estimation is small enough that it does not impact our overall findings.

The heuristic to filter out non-relevant polymorphic predicates is just an approximation, because of limitations in the static analysis. Although we do our best

to determine if the receiver of a type predicate is used later on, some cases are very hard to cover. For instance, some usages may not include a literal block, but only a variable to reference a block; this makes it impossible for our analysis to determine if the receiver is used—however, there are just 91 instances of this case in the corpus. Similarly, some relevant usages may be wrongly classified, because the predicate receiver may be used in a way in which the type information is not relevant, *e.g.* calling the same method in both branches. We conjecture these cases are negligible too.

Another threat is related to the natural language in which the analyzed projects are developed. Our isXxxx heuristic is of course only valid in English. However, the vast majority of the source code in our corpus is indeed in English. We have anecdotal evidence of projects in other natural languages, but they constitute a small minority. Further, these projects still use type predicates defined in the Smalltalk kernel, or in libraries or frameworks they use.

Internal Validity. In Section 6.3, we introduced usage categories of predicates such as Dispatch, Collections, or Assertions. These categories are in practice not entirely orthogonal though. For instance, a predicate used in Collections can also act as a Dispatch:

```
(figures anySatisfy: #isCircle) ifTrue: [self changeCircle]
```

In the case that a predicate usage is ambiguous, we add it to the category with the highest priority, *i.e.* the one which is closest to the actual usage of the predicate (in the case above, anySatisfy: takes precedence over ifTrue:, so we classify it as Collection, not Dispatch). This procedure might favor certain usage categories over others and hence influence the results for the distribution of predicate usages. However, we are interested in the closest usage context that may impact the program's control flow, which explains our choice of priorities. Further, the number of cases where there is an overlap is low; 7,573 of the 107,897 (7%) type predicate usages were found to belong to two usage contexts.

For predicate usages not following one of the main categories (*i.e.* Dispatch, Collections, Assertions, or Forward) we introduced a catch-all category. This Others category might actually also contain predicate usages of other categories. Since

we do not closely analyze the Others predicate usages in our study, we might ignore relevant usages. However, as the Other category only contains 2% of all usages, its impact on the study results is marginal. At worst, we are slightly under-estimating the relevant predicate usages.

Our analysis suffers from name clashes: in a dynamically-typed language like Smalltalk it is statically impossible to determine whether two different definitions of a predicate isCircle in different projects (or even in one single project) refer to the same concept or whether they are unrelated. We currently search in the entire corpus for predicate declarations and hence consider all definitions of isCircle as one single concept and therefore all usages of this selector as users of one single predicate. Doing a project-based analysis would have the opposite problem, finding that two usages of isCircle in different projects would be referring to two distinct predicates, even if they are genuinely related (*e.g.* one of the projects may extend the other, or use it as a library or framework).

The dynamic analysis we performed in Section 6.6.5 might be incomplete and imprecise, as the results of any dynamic analysis are highly dependent on the particular execution scenarios. For this reason, we opted to execute the test suites of the analyzed projects to maximize completeness and precision. These test suites are likely to cover the important features of the analyzed systems, so we expect the ratio of constant and variable predicates to not vary much in other scenarios.

External Validity. As we only analyze open-source projects we cannot generalize our results to close-source industrial projects. Similarly, as we only take into account projects stored in Squeaksource, contributed by Squeak and Pharo developers, we do not know whether the results would be different when analyzing code of other Smalltalk dialects such as VisualWorks.

Our corpus of analyzed projects only contains Smalltalk source code. Our assumption is that Smalltalk code, since it is free from typing constraints, is a "blank slate" in terms of how developers do dispatch based on type predicates; a language with pre-existing constraints might bias the results one way or another. However, carrying a replication of our study on another corpus (*e.g.* the Qualitas corpus of Java source code [Tempero et al., 2010]) would allow the community to

better understand the contrasts between languages and the biases introduced by a particular type discipline.

To increase the representativeness of the study, we limited the analysis to the 1,000 largest projects stored in Squeaksource. This allows us to exclude toy or experimental projects from the analysis. However, doing so might also impose a threat to external validity.

6.8 Related work

Tobin-Hochstadt and Felleisen [Tobin-Hochstadt and Felleisen, 2008] report on their practical experience porting Racket programs to Typed Racket, but do not give any empirical measurements about the prevalence of the patterns their occurrence type system supports. When introducing logical types [Tobin-Hochstadt and Felleisen, 2010] Tobin-Hochstadt and Felleisen report on a study focused on the use of some known predicates (like `number?`) as well as on the use of the `or` logical combination, which was not supported in their previous system. They report that in the source code base of Racket, `or` is used with 37 different primitive type predicates almost 500 times, as well as with user-defined predicates. These numbers justify the logic reasoning framework they propose. Our experiment further confirms that both occurrence typing and logical types are useful, in the context of object-oriented languages.

When proposing flow typing, Guha *et al.* briefly report on the prevalence of type tests and related checks across a corpus of JavaScript, Python and Ruby code [Guha et al., 2011]. In 1.5 million lines of code, they detect 13,500 occurrences of type testing operators. They use this measurement as a motivation for their work. We detect proportionally many more occurrences, even without considering user-defined polymorphic predicates (about 3 times more). Our study strengthens the argument that object-oriented programmers tend to use explicit type checks sufficiently enough to warrant specific support for them.

In the special case of non-null references, the study by Chalin and James analyzed five open-source projects, and found that 3/4 of declarations are meant to be non-null by intent [Chalin and James, 2007]. Our study does not directly measure

this, but finds that even if this is the case in our corpus, a significant number of the remaining type predicates do concern nullity.

Winther presents Guarded Type Promotion [Winther, 2011], a type system extension for Java that eliminates the need for explicit casts (called guarded casts) through analyzing type predicates, *i.e.* instanceof occurrences in control flow statements. Guarded Type Promotion uses an intraprocedural data-flow analysis to detect (only) guarded casts. Other kinds of casts, such as semi-guarded casts—*i.e.* casts in control flow statements where a polymorphic type predicate, such isShape, is checked—are not treated. Winther performed a simple static analysis to track casts in several Java projects. In total, 5.2 million LOC were analyzed, revealing more than 35,000 casts. A quarter (24.3%) of these casts are guarded casts, 23.1% are classified as semi-guarded casts, and the rest are casts not necessarily related to control flow. Additionally, Winther reports that Guarded Type Promotion was able to remove almost 95% of the guarded casts. These results suggest that a flow-sensitive typing is very useful in Object-Oriented languages.

Whiley [Pearce, 2013a,b; Pearce and Noble, 2011] is a statically-typed language that supports flow-sensitive typing and structural subtyping. Whiley programs compile directly to the JVM. Whiley's type system is sound. In the case of flow-sensitive typing, Whiley also supports union, intersections and negation types. Union types are used to capture the type of variables at meet points, intersection types are used for true branches, and negation types are required for false branches. The type system only tracks nominal type predicates.

Robbes *et al.* show that contrary to expectations, object-oriented software does evolve in ways that do not fit the object-oriented paradigm (by adding new classes), but rather corresponds to the functional design (by adding new methods) [Robbes et al., 2012]. This observation could partially explain why object-oriented programmers resort to explicit type checks, being the common approach of functional design. Further study would be required to analyze a significant sample of usages of type tests and see if they correspond to points in the application design where functional decomposition is more appropriate than the object-oriented one.

Malayeri and Aldrich perform an empirical study of the usefulness of structural subtyping in object-oriented languages [Malayeri and Aldrich, 2009]. They analyze 29 Java programs and find that nominally typed programs could benefit from

structural types, leading to more opportunities for code reuse, reduced number of runtime errors, and reduced amount of code duplication. Our study shows that Smalltalk programmers do not use structural type predicates such as `respondsTo:` as much as they use nominal ones. It has to be expected that using structural predicates would exhibit similar advantages to those reported by Malayeri and Aldrich, since they are, like polymorphic predicates, more flexible by not depending on the actual implementation hierarchy.

Beckman *et al.* study object protocols in almost two million lines of code of open-source Java programs, reporting that about 7% of the types in Java define protocols, and 13% of all classes are clients of these protocol-defining classes [Beckman et al., 2011]. Our study suggests that a large number of predicates used in practice are used to reason about the state of objects. Static reasoning on protocols and object states is directly related and the techniques used, such as typestate checking [Bierhoff and Aldrich, 2007; DeLine and Fähndrich, 2004; Strom and Yemini, 1986; Wolff et al., 2011], could be used likewise. It would be interesting to study more precisely the state-dependent predicates we identified in our experiment and see if they are related to protocol-defining classes, as this would suggest a clear potential for typestate-oriented programming [Aldrich et al., 2009; Wolff et al., 2011].

In a retrospective study of 10 open-source Java systems, Parnin *et al.* studied the adoption of Java generics by Java developers [Parnin et al., 2013]; they found that if developers do adopt generics (after a sometimes consequent delay), it is principally because a minority of developers are championing the practice. Furthermore, developers do not usually convert old code to generics. These results show that adoption of a type system, or the extension of one, is far from automatic; careful thought need to be invested in how to make the transition as painless as possible. In addition, simple, pragmatic approaches may pay surprisingly well: Parnin *et al.* found that the addition of a simple StringList class, instead of a type extension, would have covered 25% of all generic use cases.

6.9 Conclusion

Designing a type system for an existing dynamic object-oriented language is a hard task. The choice of features to include in the type system is delicate, in order to find a good compromise between coverage of existing programming idioms, strength of the guarantees brought by the type system, as well as complexity and usability of the type system. This work sheds light on the need to support explicit type-based reasoning in object-oriented programs, looking at a large Smalltalk codebase (more than 4 millions lines of codes).

Despite being shunned by good practices, type predicates do end up being present in object-oriented source code written by practitioners. The prevalence of these predicates has practical consequences in a variety of contexts. In this work, we discuss consequences on two of them: to inform practitioners of the prevalent use of type predicate in practice; and on the design of type systems that can efficiently propagate the information exposed by those predicates. The results and findings in this chapter can also contribute to the discussion of type predicates in other areas, such as those in refactoring and teaching. On the refactoring front, these results may assist practitioners when they attempt to remove usages of such predicates. On the pedagogical front, current pedagogical approaches would benefit from contrasting the core principle of the object-oriented paradigm with the state-of-the-practice, raising awareness about the typical pitfalls and design alternatives. However, we leave these analyses for future work.

We find that:

RQ1: Programmers do use a fair number of type predicates to do explicit dispatch: overall, there is a density of one such check per 50 lines of code. The problem *is* prevalent in practice. There is need for more awareness on this issue. Hence, flow-sensitive typing—in any of its possible forms—is useful for objects.

RQ2: The Nil predicate accounts for three quarters of all usages, and more than 90% of those usages are in a direct control flow statement. This suggests that a simpler, less general approach specifically tailored to this case would already enjoy a broad applicability. In other words, just the introduction of non-null types would be a very valuable help to practitioners.

RQ3: Logical combinations of type predicates are prevalent overall, though significantly more prevalent in polymorphic predicates (28.5%) than in the Nil predicate (11.2%). This result can be seen as a validation of the need for logical types in occurrence typing on the one hand, and as evidence that logical types are not as necessary when only addressing the special predicates on the other hand.

RQ4: A good proportion of type predicates are actually not constant over time. Even though a (limited) dynamic analysis lowered this proportion considerably, the results still suggest that flow-sensitive type systems should be able to deal with mutable state properly.

Relevance for Gradualtalk

This study helps us understand the use of type predicates to do explicit dispatch. We conclude that flow-sensitive typing (with support for logical combinations of type predicates) is useful for objects. In the particular case of Smalltalk, we find a frequency of one type predicate check for each 50 lines of code, where three quarters of them are actually checking object nullness. This tells us that a specific approach, such as non-null types, would have a more positive effect in Smalltalk programs with less effort in the type system implementation. However, the introduction of flow-sensitive typing in Gradualtalk may produce conflicts or undesired combinations with other features such as gradual typing, structural types and generics. Indeed, Boyland [Boyland, 2014] reports on unsoundness issues (due to the lack of progress) when introducing type tests (*aka.* type predicates) in a gradual type system with support of structural types. This is because some well-typed programs with structural type tests can get stuck at runtime where there is the presence of ambiguity in erased selection. Hence more research is needed for understanding the impact of flow-sensitive typing in Gradualtalk. Moreover, we consider that the prevalence of the type-based dispatch idiom is still too dispersed to be considered a main feature. Hence the implementation effort may be too high in comparison with other features, such as union types. We therefore delay its support in Gradualtalk until the next version. This is because, in the first version of Gradualtalk, we primarily focus on main features and idioms of Smalltalk.

Part II: Gradualtalk

Chapter 7

Introduction to Gradualtalk

In this chapter, we showcase the features of the language using clear and concrete examples, *i.e.* code snippets. Initially, we introduce gradual typing (Section 7.1) and lambdas (Section 7.2) as core features. We then present the typing features implemented in the first version of Gradualtalk[1]. We start with self types and its interaction with meta classes in Section 7.3. We then continue with explicit coercions (Section 7.4). Section 7.5 introduces generic types, also know as parametric polymorphism. We then present union types in Section 7.6. Section 7.7 presents nominal and structural types separately, and then in a combined form. The last introduced feature is incremental type checking in Section 7.8. Finally, we present the specific set of typing rules for Gradualtalk and discuss its safety and type soundeness properties (Section 7.9).

7.1 From dynamically typed to gradually typed code

A developer is trusted with the development of the geometric calculation module for a graphics application. She starts writing dynamically-typed code. The following code snippets are the implementation of two example methods: euclidean distance and a class method for creating points.

[1] Available at http://www.pleiad.cl/gradualtalk

```
Point≫ distanceTo: p
   |dx dy|
   dx := self x – p x.
   dy := self y – p y.
   ↑ (dx squared + dy squared) sqrt
```

```
Point class≫ x: aNumber1 y: aNumber2
    ↑self new x: aNumber1; y: aNumber2
```

After development and testing, the developer wants to increase robustness and provide basic (checked) documentation for these methods. For that purpose, she needs to type the method declarations of those methods. The following example is the typed version of the method distanceTo:.

```
Point≫ (Number) distanceTo: (Point) p
   |dx dy|
   dx := self x – p x.
   dy := self y – p y.
   ↑ (dx squared + dy squared) sqrt
```

The method declaration of this method specifies that the type of the parameter p is Point, while the return value type is Number. Because the local variables dx and dy are not annotated, they are treated as being of type Dyn, *i.e.* the type of any object in a dynamically-typed language.

Note that the Dyn type is also very helpful to type methods that cannot be otherwise typed precisely, either because of a limitation of the type system, or because of inherent dynamicity. The typical example of the latter is reflective method invocation, done in Smalltalk with the perform: method:

```
Object >> (Dyn) perform: (Symbol) aSymbol
```

The argument to perform is a Symbol, which denotes the name of the method (selector) that must be invoked on the receiver object. In general, the return type cannot be statically determined. Declaring it as Dyn instead of Object means that clients of this method can then conveniently use the return value at any type, instead of having to manually coerce it. Hence, the use of Dyn implies an untyped code, while the use of Object (or any other type) implies a typed code.

7.2 Closures

The next method to type in our example is perimeter:. This method takes as parameter a closure that computes the distance between two points, and returns the value of the perimeter of the polygon, using the provided closure. Closures, also known as blocks, are a basic feature in Smalltalk, so the type system supports them. The following code is the typed version of the perimeter: method declaration:

```
Polygon ≫ (Number) perimeter: (Point Point → Number) metricBlock
   ...
```

In the example, the parameter metricBlock is a closure; its type annotation specifies that it receives two Points and returns a Number.

7.3 Self and metaclasses

The next method to type is y:. This method is a setter for the instance variable y. Its return value is self, the receiver of the message. The following code corresponds to its typed method implementation:

```
Point≫ (Self) y: (Number) aNumber
   y := aNumber.
```

Self is the type of self, as in the work of Saito *et al.* [Saito and Igarashi, 2009]. Declaring the return type to be Point would not be satisfactory: calling y: on an instance of a subclass of Point would lose type information and forbid chained invocations of subclass-specific methods.

We now consider the class method x:y:, which acts as a constructor:

```
Point class≫ (Self instance) x: (Number) aNumber1 y: (Number) aNumber2
   ↑self new x: aNumber1; y: aNumber2
```

Self instance is the type of objects instantiated by self. Self instance is therefore only applicable when self is a class or metaclass. This was inspired by the type declaration "Instance" in Strongtalk [Bracha and Griswold, 1993]. Using Self instance instead of Point brings the same benefits as explained above. Constructor methods are inherited, and Self instance ensures that the returned object is seen as an object of the most precise type. The dual situation, where an object returns

the class that instantiated it, is dealt with using Self class, which is also inspired by Strongtalk.

Self instance in Gradualtalk and Strongtalk Instance are similar, but subtly different. The difference shows up when looking at the Class class, and related classes. Recall that in Smalltalk, classes are objects, instance of their respective metaclass, which derive from the Class class. The problem is that in Strongtalk, inside that class, the type Instance is a synonym of Self. This means that all methods defined in Class—and its superclasses ClassDescription and Behavior—lack a way to refer to the type of their instances. This limitation can be observed in several places. For example, the return type of Behavior ≫ #new is Object in Strongtalk, which is imprecise, while it is Self instance in Gradualtalk. To type the method new correctly, Strongtalk needs to redefine new in the subclass Object class (the metaclass of Object), and change its return type to Instance. Another example of this problem is in the following method from Behavior:

```
Behavior≫ (Self) allInstancesDo: (Self instance → Object)aBlock
    "Evaluate the argument, aBlock, for each of the current instances of the  receiver."
```

Using Self instance above as the argument type of the block denotes any possible instance of a Behavior object. Properly typing this method is not possible in Strongtalk: as a consequence, it has been moved down the hierarchy to the Object class class. Self types in Gradualtalk are strictly more expressive than in Strongtalk.

7.4 Casts

The following code is the method perimeter, which computes the perimeter using the euclidean metric:

```
Polygon ≫ perimeter
  ↑ self perimeter: [:x :y| x distanceTo: y]
```

This dynamically-typed method invokes the perimeter: method with a (Dyn Dyn → Dyn) closure, yet this method expects a (Point Point → Number) closure. In the type system of Gradualtalk, the former closure type is *implicitly* cast to the latter. As a result, the developer does not need to write any type annotation.

The language also gives the programmer the option of explicitly coercing from one type to another type. An explicit cast is shown in the following:

```
Polygon ≫ perimeter
  ↑ self perimeter: [:x :y| (<Integer> x distanceTo: y)]
```

The return value of the expression "x distanceTo: y" is cast to an Integer. If it is not an Integer at runtime, a runtime exception is raised.

7.5 Parametric polymorphism

Consider the following piece of code, where an array of Dyn objects is defined:

```
|(Array) points|
...                 "filled with points"
(points at: 1) x    "potentially unsafe"
```

The programmer knows that any element of the array is a Point, and invokes the method x of class Point. Sadly, the type system cannot guarantee a safe method call at compile time, consequently a coercion is introduced by the type system. Here, the type information is lost, forcing the programmer to either use casts or the Dyn type. Casts need to be manually inserted, which is cumbersome and error prone.

To solve this problem, Gradualtalk supports parametric polymorphism [Pierce, 2002]. Adding parametric polymorphism to gradually typed languages is not new: Ina and Igarashi [Ina and Igarashi, 2011] presented a formalization and initial implementation of generics for gradual typing in the context of Featherweight Java. We adopt their approach in Gradualtalk. As of now, generics are implemented using type erasure as in Java. This is because of simplicity in the implementation and compatibility with compiled code. Nevertheless, we are considering adding reification for type parameters and arguments in a future version of Gradualtalk.

Gradualtalk includes a generically-typed version of the Collection library. For instance, the next piece of code solves the above problem by introducing Point as a type argument to the generic Array type:

```
|(Array<Point>) points|
...
(points at: 1) x    "safe call"
```

Below is an example of a generic method definition:

```
Collection<e> ≫ (a) add: (a <: e) newObject
```

This method inserts an object in a collection. Interestingly, in Smalltalk, the return value of this method is the added object. Therefore, in order to not lose type information, we use a bounded type variable a, subtype of the collection element type e, and specify a as the return type. Note that by convention, in Gradualtalk, type variables are single lowercase characters, similar to Haskell and ML.

Along with generics the type system also supports polymorphic functions (blocks in Smalltalk), which is useful in several cases, *e.g.* higher-order functions in collections:

```
Collection<e> ≫ (Self<f>) collect: (e → f) op
Collection<e> ≫ (Self<e>) select: (e → Boolean) pred
Collection<e> ≫ (f) inject: (f) init into: (f e → f) op
```

Note that the two first methods above use parametric self types to precisely type their return values.

Combining parametric types with some other typing features may produce new and interesting properties. For instance, the interactions between the Dyn type and generics, called bounded dynamic types [Ina and Igarashi, 2011], permits flexible bounded parametric types. Gradualtalk does not include this feature as of now, because we have not found conclusive evidence in practice that justifies it yet. Another interesting interaction occurs between self types and generics, called self type constructors [Saito and Igarashi, 2009], allowing programmers to parametrize self types. Self type constructors are required to properly type collections in Gradualtalk, and are therefore supported.

7.6 Union types

The following piece of code is a polymorphic implementation of the ifTrue:ifFalse: method, where we use RT as a placeholder for the return type:

```
Boolean≫ (RT) ifTrue: (→ a) trueBlock ifFalse: (→ b) falseBlock
  ...
```

The method receives two block arguments, one for the true case named trueBlock, and one for the false case, named falseBlock. In this example each block can evaluate to a result with a different type. Because of this the trueBlock has return type a and the falseBlock has b, and a=b is not always the case. Consequently, at this moment there are two possible values for RT:

Object. The type Object does not provide any information to programmers. Even if we consider the lowest common ancestor between types A and B, still some type information is lost. Therefore, programmers are forced to insert a cast to get the real type.

Dyn. We get the flexibility that we need, but again type information is lost.

While this is a simple example, there are several places in the corpus where examples like this can be found. To solve this problem, we use *union types* [Pierce, 2002]. These allow programmers to join several types in a single one, via disjunction. Union types are represented by | in Gradualtalk. A union between types a and b solves the problem of the example, letting the programmer specify that only one of these is possible.

```
Boolean≫ (a | b) ifTrue: (→ a) trueBlock ifFalse: (→ b) falseBlock
```

Another interesting example is the following method:

```
Collection<e> ≫ (Self | a) ifEmpty: (→ a) aBlock
  ↑ self isEmpty ifTrue: [ ↑aBlock value ] ifFalse: [ self ]
```

The method returns the result of the invocation of aBlock (of type a) if the collection is empty, or self otherwise. To type this precisely, a union type Self | a is used.

When using a variable typed with a union type a | b, the programmer can safely call common methods in a and b. Calling specific methods of a or b requires explicit disambiguation, for instance using isKindOf: to perform a runtime type check and then using a coercion. Flow-sensitive typing may be useful in those cases to avoid having to explicit coerce values after a runtime type check, however this feature is not supported in the first version of Gradualtalk.

7.7 Structural and nominal types

In this section, we introduce two important typing features, structural and nominal types. Although these kinds of types seem contradictory, they are actually complementary. We first present them separately, then in a combined form.

7.7.1 Structural types

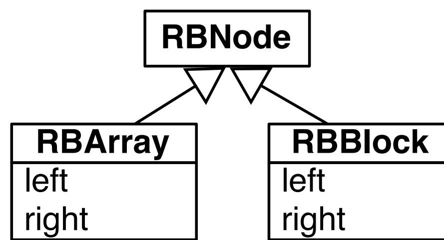

Figure 7.1: A common structural protocol.

Figure 7.1 describes the RBNode hierarchy (RB is shorthand for Refactoring Browser) that represents abstract syntax tree nodes in a Smalltalk program. In the example, only the classes RBArray and RBBlock understand the selectors left and right.

Consider the following code that is added to handle brackets in the parser, where we use AT as a placeholder for the argument type:

```
RBParser ≫ bracketsOfNode: ( AT ) node
... node left.
... node right.
```

Consequently, there are three possible values for AT:

RBNode. RBNode is the common ancestor of RBArray and RBBlock. However any call to the methods left and right will be rejected by the type system, because RBNode does not define these methods. Even a cast will not help, because the type system cannot statically determine if either RBArray or RBBlock will be the correct type.

RBArray | RBBlock A union type could be a good solution. However it is not scalable if more nodes include brackets later on in development.

Dyn. The code will be accepted by the type system, but again type information is lost.

This problem appears because RBArray and RBBlock have no relation between them except from being nodes, and not all nodes have brackets. But RBArray and RBBlock also share a set of common methods used in the method bracketsOfNode:. Therefore, objects of type AT will understand this set of methods, *i.e.* the selectors left and right. A type with this structural representation, *i.e.* set of method types, is called a *structural type* [Pierce, 2002]. This means that the type system permits as argument any object that understands the selectors in the structural definition. Using structural types, the solution is as follows:

```
RBParser ≫ bracketsOfNode: ({left (→ Integer) . right (→ Integer)}) node
```

The use of structural types allows programmers to explicitly specify a set of methods that an object must implement. These methods are the only available methods for the structurally-typed variable (node, in the above example) and therefore any call to another method will be invalid, unless a cast is used.

Type Alias The verbosity of structural types could be a problem for programmers, and even worse it can lead to an agglomeration of anonymous protocols. To solve this, Gradualtalk permits the use of a *type alias* [Pierce, 2002], where programmers can give names to arbitrary types, in order to enhance readability. Note that the use of a type alias is not only restricted to structural types, for example Nil is a type alias for the UndefinedObject type.

Named Protocols Smalltalk does not support explicit interfaces or protocols. Instead, programmers rely on their understanding of what a given protocol is, and provide the necessary methods. For example consider the *pseudo* protocol "property", where the methods that handle properties in an object are listed:

```
propertyAt:
propertyAt:Put:
propertyAt:ifAbsent:
propertyAt:ifAbsentPut:
removeProperty:
removeProperty:ifAbsent:
```

Not making this protocol explicit is fragile, because it may evolve over time.

By combining a structural type and a type alias, programmers are able to define *named protocols*, which are similar to nominal interface types, except that they are checked structurally. With this, protocols are explicitly documented, and programmers can explicitly require them *e.g.* as an argument type of a method, without losing the flexibility of structural typing. The previous protocol can be declared as the following type:

```
PropertiesHandler := {
  propertyAt: (Symbol → Dyn) .
  propertyAt:Put: (Symbol Dyn → Dyn) .
  propertyAt:ifAbsent: (Symbol (→ Dyn) → Dyn) .
  propertyAt:ifAbsentPut: (Symbol (→ Dyn) → Dyn) .
  removeProperty: (Symbol → Dyn) .
  removeProperty:ifAbsent: (Symbol (→ Dyn) → Dyn)
}
```

Note that a named protocol can serve to give a type to a trait [Schärli et al., 2003]. However, traits come with a specific implementation, while named protocols are pure interface specifications. The same protocol can be implemented by different traits.

7.7.2 Nominal types

Nominal types [Pierce, 2002] are the types that are induced by classes, *e.g.* an instance of class String is of type String. One of the primary advantages of nominal types is to help programmers to easily express and enforce design intent. Because of this, most mainstream statically typed object-oriented languages support nominal types rather than other alternatives, such as structural types.

Nevertheless, structural types offer their own advantages [Malayeri and Aldrich, 2008, 2009]. For instance, structural types are flexible and compositional, providing better support for unanticipated reuse. This is because they imply a more flexible subtyping relationship compared to nominal subtyping, allowing unrelated classes in the class hierarchy to be subtypes. Taking this into account, some type systems [Doligez et al., 2011; Malayeri and Aldrich, 2008; Siek and Taha, 2007] use structural types. In fact, a nominal type also can be considered in terms of its

structural representation. This means that instances of class String have a structural type: the set of all methods that a string understands. Using such a type alias, programmers can benefit from the advantages of structural types.

In the case of Smalltalk, considering class-induced types as their structural representation is, however, not suitable. This is because Smalltalk classes tend to have a large number of methods, which makes it impractical to comply with subtyping outside of the inheritance hierarchy. For instance, consider the SequenceableCollection class, which has hundreds of methods. If the programmer wants to define a subtype that is not a subclass she must implement all methods in the SequenceableCollection class. A solution is to combine structural and nominal types, as discussed next.

7.7.3 Reconciling nominal and structural types

Figure 7.2 describes the hierarchy of some classes in Smalltalk that define the selectors left and right with type signature (→ Integer). With this new set of classes, the solution presented in Section 7.7.1 is not complete. This is because the type system will accept calls to the method bracketsOfNode: with a parameter that complies with the protocol, *e.g.* a Morph object, but which is not a node.

Gradualtalk supports the combination of nominal and structural types, similar to Unity [Malayeri and Aldrich, 2008] and Scala [de Lausanne , EPFL].[1] A type combines both a nominal part and a structural part, as in A{m1...mn}. For instance, consider the following modification in the parameter type that takes structural and nominal types into account:

```
RBParser ≫ bracketsOfNode: (RBNode{left (→ Integer) . right (→ Integer)}) node
```

Here the type system is requesting an explicit RBNode object that has selectors left and right. Now a call with a Morph object as argument is rejected because it is not an RBNode.

[1]Note that because Scala compiles to the JVM, structural invocations introduce an extra performance penalty due to reflection. Gradualtalk, on the other hand, does not penalize structural invocations.

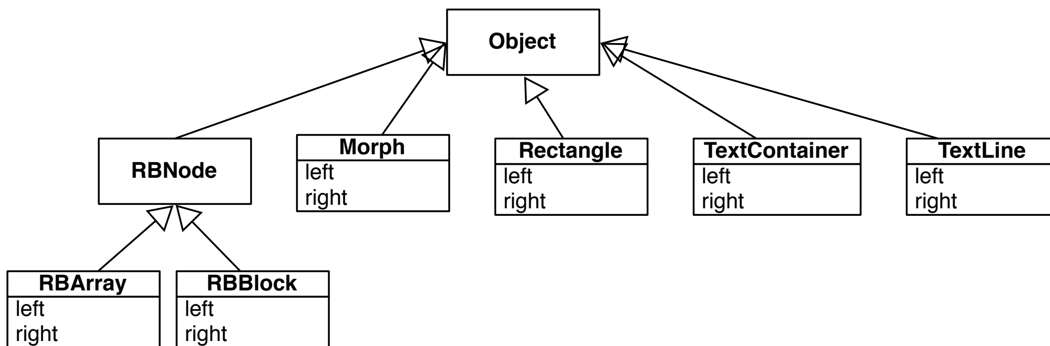

Figure 7.2: A common structural protocol across projects.

Note that a nominal type A is a syntactic shortcut for the combined type A{} (empty structural component), while a structural type {m1, m2, ...} is the equivalent of Object{m1, m2, ...}.

Flexible Protocols Interestingly, the combination of the Dyn type with a structural type produces a *flexible protocol*, of the form Dyn {m1, m2, ...}. A flexible protocol represents objects that must comply with a protocol (structural part), but can otherwise be used with an implicit coercion (Dyn part). Consider the following piece of code:

```
Canvas≫ (Self) drawPoint:(Dyn {x(→ Integer). y(→ Integer)}) point
... point x. "safe call"
... point y. "safe call"
... point z. "not an error, considering point as Dyn"
```

The last statement does not raise a type error, because point has been typed with a flexible protocol newinstead it insert an implicit casts. However, calling drawPoint: with an argument that does not support the {x, y} protocol is a static type error. Hence, flexible protocols allow programmers to represent flexible structural types without the penalty of having to use explicit casts. Since Unity and Scala are not gradually-typed, flexible protocols are a novel feature of Gradualtalk.

7.8 Live system

Nearly every Smalltalk environment is a live system. This means that the developer writes the code, runs it and debugs it in the same execution environment. To

support this live environment, individual methods can be compiled and added to an existing class. This is in contrast to other languages where the smallest compilation unit is a class. This feature of a live environment raises three problems for a typechecker.

The first problem is the granularity of the compiling process. In Smalltalk, the compilation process is done per method, instead of per class. Traditionally, a type checker prevents compilation when type errors are found. But with such a fine-grained compilation process, the traditional approach does not work. For example, if a programmer needs to define two mutually-dependent methods, when the first method is defined, the typechecker cannot know if the second method referenced is going to be defined later. The error should however not block the programmer from keeping this as-yet-buggy method and then define the second method. The same situation happens when loading code, since code loading in Smalltalk is just a script adding definitions one-by-one. In order to address this issue, we decouple the typechecking process from the compiling process: Gradualtalk can compile methods with type errors. Errors are collected in a separate typing report window.

The second problem is that the work done by the typechecker can become obsolete when new methods are introduced or an old method is modified. For example, if the return type of a method is changed from Integer to String, all methods that invoke it can potentially become ill-typed. To solve this problem, we introduce a dependency tracking system based on Ghosts [Callaú and Tanter, 2013], which allows the type system to properly support partially-defined classes and circular dependencies. Undefined classes and methods that are referenced are considered as ghost entities, about which type information is gathered. This allows the type system to check for consistent usage of as-yet-undefined entities. Dependency tracking considers both defined entities and ghosts. Each time the programmer updates or deletes definitions, the dependency tracker notifies the type system of which methods must be checked again. In case the type system detects some type errors, it reports the exact points of failure. More precisely, the dependency tracking system records bi-directional references between *dependents* and *dependees*. These dependencies are updated whenever a method is type-checked, and whenever the format of a class definition (variables) changes. The result of this process is a dependency graph of dependent and dependee nodes. A dependent node is

either a pair $(class, selector)$, for dependent methods, or a pair $(class, variable)$, for dependent instance or class variables. A dependee node is either a class, for type related dependencies, or a pair $(class, selector)$ for method invocation dependencies. Whenever a dependee node is updated, all dependents are re-checked and re-compiled (necessary because implicit cast insertion may have to change).

The third problem occurs when compiling typed system code that is critical. It is common that programmers commit errors when typing code, especially if it was not developed by them. In normal code, it is not a problem that a method fails when compiling, or cast errors are raised when they are executed. However, in critical code, having cast errors is fatal. For example, if the default error handler raises a cast error, an infinite loop is produced and the system is irresponsive, making it impossible to use the debugger. To address this problem, in Gradualtalk runtime casts insertion and checking can be disabled or enabled at will. This means that programmers can enable the type system for some classes, and disable for others. To gradually type important and critical system parts, we used this feature to first focus on debugging the cause of typecheck errors at compile time, then progress to runtime cast errors. Also, disabling runtime casts after a cast error is raised allows us to use the debugger without further interference of the type system.

Gradual or optional? Disabling runtime casts insertion was built in Gradualtalk to address the problem discussed above. Interestingly, it can also be used to make the type system of Gradualtalk an *optional* type system, just like that of Strongtalk. Moreover, because code instrumentation can be enabled or disabled at will, Gradualtalk allows optionally-typed and gradually-typed code to co-exist in the same system; a combination which, to the best of our knowledge, has not been explored so far.

7.9 Gradualtalk static semantics

In this section, we introduce and discuss an early formalization of Gradualtalk. Gradualtalk is a type system based on a well-known and strong theoretical basis from gradual typing and each of the typing extensions presented in this chapter.

Hence, we only present a set of specific typing rules of the novel interactions in Gradualtalk. Additionally, we discuss the type safety and soundness property of Gradualtalk.

7.9.1 Types in Gradualtalk

τ	$::= \gamma \mid \overline{\tau} \rightarrow \tau \mid \tau + \tau \mid \gamma\text{<}\overline{\tau}\text{>} \mid \gamma\sigma$	Type
γ	$::= \nu \mid \epsilon \mid \mathsf{x} \mid \mathsf{Dyn}$	Ground type
ν	$::= \mathsf{C} \mid \mathsf{C\ class} \mid$	Nominal type
ϵ	$::= \mathsf{Self} \mid \mathsf{Self\ instance} \mid \mathsf{Self\ class}$	Self type
σ	$::= \{\overline{\mathsf{m}\ \overline{\tau} \rightarrow \tau}\}$	Structural type

Table 7.1: Types in Gradualtalk

Table 7.1 presents the grammar of types in Gradualtalk. C ranges over class names in the system, x ranges over type variables and m ranges over selector names. A bar over a type term denotes zero or more occurrences of the term.

A type τ is a either a ground type γ, a function type, a union type, a generic type, or a combined type with a structural component σ. A ground type is either a nominal type ν, a self type ϵ, a type variable or Dyn. A structural type σ is a list of selector types, including a selector name and a function type.

7.9.2 Self types rules

Although the semantics of the type Self are well known, this is not the case for the Self instance and Self class types. To define them properly, we define the concepts of instance types and class types. The instance type of τ is the type of objects instantiated by objects of type τ. If an object of type τ cannot have instances, then the instance type of τ is undefined. The class type of τ is the type of the class object that produces objects of type τ. Figure 7.3 and Figure 7.4 define the rules for instance types and class types respectively. Note that a key challenge in Smalltalk is to properly take into account the core classes that describe classes and metaclasses: Behavior, its subclass ClassDescription, and its subclasses Class and Metaclass.

instance(Nil)=Nil

instance(C class)=C

instance(Metaclass)=Class

instance(A)=Object, if Class <: A <: Behavior

instance($\mathsf{Self_C}$ class)=$\mathsf{Self_C}$

instance($\mathsf{Self_C}$)=$\mathsf{Self_C}$ instance, if C <: Behavior

instance($\mathsf{Self_C}$ instance)=instance(instance(C))

instance($\gamma\sigma$)=instance(γ)

instance($\tau_1 + \tau_2$)=instance(τ_1)+instance(τ_2)

instance(x)=instance(upperbound(x))

instance($\gamma<\overline{\tau}>$)=instance(γ)

instance(Dyn)=Dyn

Figure 7.3: Definition of the instance relation on types.

A self type can be found in a calling context, as the type of a parameter or return value of an invoked method, or in called context, as the type of a variable or of the return value. In a calling context, self types are replaced by the type of the receiver. If the type of the receiver of the invoked method is τ, Self, Self instance and Self class are replaced by τ, instance(τ) and class(τ) respectively. In a called context, the type Self is represented in the type system as $\mathsf{Self_C}$, where C is the current class. Self instance and Self class are represented in the same manner.

7.9.3 Subtyping

One important feature in object-oriented languages is subtyping, by which an object of a given type can also be considered as being of any of its supertypes. The presence of several kind of types in Gradualtalk makes the subtyping relationship nontrivial. We next explain how it is treated.

$\text{class}(\mathsf{Nil}){=}\mathsf{Nil}$

$\text{class}(\mathsf{Object}){=}\mathsf{Behavior}$

$\text{class}(\mathsf{C}){=}\mathsf{C\ class}$, if $\mathsf{C} \not<: \mathsf{Behavior} \land \mathsf{C} \neq \mathsf{Object}$

$\text{class}(\mathsf{A}){=}\mathsf{A}$, if $\mathsf{A} \in \{\mathsf{Behavior},\mathsf{ClassDescription}\}$

$\text{class}(\mathsf{Class}){=}\mathsf{Metaclass}$

$\text{class}(\mathsf{Metaclass}){=}\mathsf{Metaclass\ class}$

$\text{class}(\mathsf{Self_C\ class}){=}\text{class}(\text{class}(\mathrm{C}))$

$\text{class}(\mathsf{Self_C}){=}\mathsf{Self_C\ class}$

$\text{class}(\mathsf{Self_C\ instance}){=}\mathsf{Self_C}$

$\text{class}(\gamma\sigma){=}\text{class}(\gamma)$

$\text{class}(\tau_1 + \tau_2){=}\text{class}(\tau_1){+}\text{class}(\tau_2)$

$\text{class}(\overline{\tau} \to \tau){=}\mathsf{BlockClosure\ class}$

$\text{class}(\mathsf{x}){=}\text{class}(\text{upperbound}(\mathsf{x}))$

$\text{class}(\gamma{<}\overline{\tau}{>}){=}\text{class}(\gamma)$

$\text{class}(\mathsf{Dyn}){=}\mathsf{Dyn}$

Figure 7.4: Definition of the class relation on types

Basic Forms of Subtyping Lambda types, self types, union types and parametric types have well-known subtyping relationships Cardelli [1997b]; Ina and Igarashi [2011]; Pierce [2002]; Saito and Igarashi [2009]. Gradualtalk follows these rules. However, because of our extension to self types, there are two additional subtyping rules concerning self types:

$$\text{(Self instance)}\ \frac{}{\mathsf{Self_C\ instance} <: \text{instance}(\mathsf{C})}$$

$$\text{(Self class)}\ \frac{}{\mathsf{Self_C\ class} <: \text{class}(\mathsf{C})}$$

Bottom Type In Gradualtalk Nil (which is an alias for UndefinedObject) serves as the bottom type. Since this type is a subtype of any other type, the programmer can use either nil or raising exceptions in any place where a typed object is expected.

Nominal and Structural Subtyping As explained in Section **??**, Gradualtalk supports the combination of nominal and structural subtyping as in Scala. First, note that Gradualtalk (as most mainstream languages) equates nominal subtyping with the inheritance relationship. Subtyping of mixed types is described by the following rule:

$$\text{(Mixed)}\frac{\gamma_1 <: \gamma_2 \qquad \text{structural}(\gamma_1) \cup \sigma_1 <: \sigma_2}{\gamma_1\sigma_1 <: \gamma_2\sigma_2}$$

This rule states that a mixed type A {n1,...} is subtype of B {m1,...} if and only if A is a nominal subtype of B and the *union* of {n1,...} and all the methods of A (ie. the structural view of A) is a structural subtype of {m1,...}. The definition of structural(.) is direct and omitted here for brevity.

Consistent subtyping Gradual typing extends traditional subtyping to consistent subtyping *consistent subtyping* Siek and Taha [2007]. Consistency, denoted $\sim$, is a relation that accounts for the presence of Dyn: Dyn is consistent with any other type and any type is consistent with itself. The consistency relation is not transitive in order to avoid collapsing the type relation Siek and Taha [2006]. A type τ_1 is a consistent subtype of τ_2, noted $\tau_1 \lesssim \tau_2$, iff either $\tau_1 <: \sigma$ and $\sigma \sim \tau_2$ for some σ, or $\tau_1 \sim \sigma$ and $\sigma <: \tau_2$ for some σ. The type system of Gradualtalk therefore operates based on the consistent subtyping relation.

Consistent subtyping does not present any specific challenge with respect to the different kinds of types in Gradualtalk. An interesting case to mention though is that of flexible protocols, since these are a novelty of Gradualtalk. Recall that a flexible protocol is a type of the form Dyn {m1,...}, ie. a type that combines Dyn with a structural type. The consistent subtyping relation for flexible protocols is defined by rules Mixed-Dyn1 and Mixed-Dyn2:

$$(\text{Mixed-Dyn1})\frac{\tau \lesssim \sigma}{\tau \lesssim \mathsf{Dyn}\ \sigma} \qquad (\text{Mixed-Dyn2})\frac{}{\mathsf{Dyn}\ \sigma \lesssim \tau}$$

Mixed-Dyn1 states that τ is a consistent subtype of $\mathsf{Dyn}\ \sigma$, if τ is a consistent subtype of σ. This rule makes explicit that τ must comply with the structural part of the flexible protocol. Mixed-Dyn2 states that $\mathsf{Dyn}\ \sigma$ is a consistent subtype of any τ. Indeed, it is valid to pass a value of type $\mathsf{Dyn}\ \sigma$ anywhere, since this is already the case with Dyn alone. Interestingly, both rules Mixed-Dyn1 and Mixed-Dyn2 correspond to two of the basic rules of consistent subtyping, $\tau \lesssim \mathsf{Dyn}$ and $\mathsf{Dyn} \lesssim \tau$, generalized to mixed types. Both of these basic rules are obtained when σ is the empty structure.

Note that flexible protocols also enjoy a direct subtyping relation as defined by the following rule Mixed-Dyn-sub:

$$(\text{Mixed-Dyn-sub})\frac{\sigma_1 <: \sigma_2}{\mathsf{Dyn}\ \sigma_1 <: \mathsf{Dyn}\ \sigma_2}$$

Mixed-Dyn-sub states that $\mathsf{Dyn}\ \sigma_1$ is a subtype of $\mathsf{Dyn}\ \sigma_2$, if σ_1 is a subtype of σ_2. This rule is the generalization of the reflexive rule $\mathsf{Dyn} <: \mathsf{Dyn}$ to mixed types; that rule can be recovered by considering both σ_1 and σ_2 empty.

7.9.4 Safety and type soundness

Gradualtalk is based on Smalltalk, which is a *safe* language: sending an unknown message to an object is a trapped error that results in a MessageNotUnderstood exception, instead of producing unspecified result or system crash. Gradualtalk inherits this safety property.

With respect to type soundness, Gradualtalk follows the foundational work on gradual typing by Siek and Taha [Siek and Taha, 2007]. Gradualtalk guarantees that, if a runtime type error occurs (that is, a MessageNotUnderstood exception is thrown), it is either due to an explicit cast that failed, or the consequence of passing an inappropriate untyped value to typed code.

Chapter 8

Gradualtalk Validation

In this chapter, we present the result of the validation of Gradualtalk using a corpus of seven existing projects. First, in Section 8.1 we present the corpus and the methodology. We then present the quantitative results in Section 8.2. Afterward, we present the qualitative results of the validation, in the form of bugs and optional refactoring (Section 8.3), interesting typed methods (Section 8.4) and challenging methods to type (Section 8.5). In Section 8.6, we present the threats to validity of this study. We end with a detailed conclusion in Section 8.7 on the overall validation, with a focus on the quantitative and qualitative results.

8.1 Corpus and methodology

Corpus. The corpus we study is composed of seven projects: Kernel, Collections, Gradualtalk, Ghosts, AST-Core, Zinc and Spec. In total, we typed 137 classes and 3,382 methods, which correspond to 18,780 lines of code. Kernel and Collections are both sub-projects of Pharo Smalltalk. The first provides the basic classes of Smalltalk, *e.g.* Object, Class, Integer, ClassDescription, Behavior, etc. The second set of classes we typed are the fundamental classes of the Collections framework in Smalltalk: Collection and SequenceableCollection. Gradualtalk is the implementation of the type system described in this thesis. Ghosts [Callaú and Tanter, 2013] is an IDE tool for supporting incremental programming through automatic and non-intrusive generation of code entities based on their usage. AST-Core is a set

of classes that allows programmers to produce abstract syntax trees of Smalltalk methods. Zinc is a framework that implements the HTTP networking protocol. Finally, Spec is the new standard framework to declaratively specify user interface components in Pharo.

Kernel, Collections and AST-Core were included in the corpus because of their maturity. Moreover, Kernel is a challenging package because it contains the core classes of the system and Collections are a typical benchmark for type systems. Gradualtalk, Zinc, Spec and Ghosts were included because these are libraries and tools that we are familiar with.

Methodology. The typing process was performed by four Smalltalk developers: two graduate students and two professors. Each person typed only the projects for which he was familiar. Additionally, each project was typed just by one person, however the overall typing was discussed in group. This means that for each typed project we discussed complex types, bugs, challenges and refactored code. Refactoring is not necessary in Gradualtalk due to the dynamic type, however in some scenarios a small refactoring is useful for providing more type information (see Section 8.3). Small refactoring is allowed in the study. The typing criterion was to introduce types up to a reasonable level of complexity, *e.g.* avoiding cases like large union types (Character | Number | String), and excessive use of parametric types (Array<c> (c → a) (→Dictionary<a, b>) → b). For those complex cases a more generic (*e.g.* a supertype) or a flexible type was used (*e.g.* using Dyn). We typed variable declarations (*i.e.* temporary, instance side and class side variables), method arguments and return, and block arguments. Additionally some expressions require explicit casts. We start by adding nominal and self types where possible. We then use parametric, union or structural types (and even Dyn) where more flexibility is required. Finally, to test that our types are introduced correctly, we call the type checker on a per-method basis. If the type method was accepted by the type system, we continue with the next one, otherwise, we fix the typing errors based on the error reports.

Projects	Classes	Methods	Dyn types	Non-Dyn types	Percentage Dyn types	Typed LOC	Total LOC
Gradualtalk	14*	294	69	657	9.50%	1,116	3,006
Ghosts	9	112	15	203	6.88%	338	338
AST-Core	17	579	402	1,175	25.49%	2,335	2,339
Zinc	41*	452	192	749	20.40%	1,733	1,833
Collections	2*	305	113	1,955	5.46%	2,596	16,292
Kernel	16*	1,290	652	3,439	15.94%	9,319	24,161
Spec	38*	350	174	776	18.31%	1,343	2,913
Total	137	3,382	1,617	8,954	15.30%	18,780	50,882

Table 8.1: Projects typed with Gradualtalk, * indicates these are not all classes of the project.

8.2 Overview of findings

Table 8.1 presents a quantification of the corpus composition and how much they are dynamically typed. The measure we use to calculate "how much classes are dynamically typed" is the percent of Dyn types present in type annotations in the classes compared with the total. Using this measure, the most statically-typed project is Collections (5.46% of Dyn), in contrast, the project that is most dynamically typed is AST-Core (25.49% of Dyn). The Collection framework is a well-studied case that is mostly typable when parametric polymorphism is supported. Although we only typed two classes in Collections, these two classes are very large and represent 15.9% of all LOC in Collections. Furthermore, the typed classes are the core classes of Collections, and in fact other classes share the same or similar protocol. AST-Core has a significant portion of Dyn, because several AST-Core classes use classes and methods in the Smalltalk image that are not typed yet, such as the Parser and the AST-Semantic package. Similar to AST-Core, Zinc and Spec have a high percentage of Dyn (20.4% and 18.31%) because of their dependencies to untyped code, such as network and UI respectively. In the case of Gradualtalk and Ghosts, they are projects more or less self-contained with few dependencies to untyped code. Kernel has a significant percentage of Dyn (15.9%). A possible reason for this is that typed code has dependencies to untyped code in the same package. In fact only 38.6% of Kernel is typed. The difference between these numbers reflects different stages of a migrating untyped to typed

code. In other words, the dependency of typed projects to untyped projects is the main reason of having significant percentages of Dyn in our corpus. This means that the presence of Dyn is high when starting to migrate untyped to typed code, however as long as more code is typed, Dyn percentages will decrease. However a fully typed version (*i.e.* the total absence of Dyn) of a Smalltalk program is not yet possible in Gradualtalk. This is because some libraries (or parts of them) rely on Dyn to be accepted by the type system, *e.g.* reflection API.

Projects	Parametric type		Union type		Structural type		Structural-nominal	
	cls	mth	cls	mth	cls	mth	cls	mth
Gradualtalk	0	0	0	0	0	0	0	0
Ghosts	0	0	2	3	1	1	0	0
AST-Core	4	8	0	0	1	2	1	1
Zinc	1	3	4	8	3	20	0	0
Collections	2	197	2	39	1	1	1	1
Kernel	6	36	4	15	5	17	1	1
Spec	4	115	7	17	1	1	0	0
Total	18	359	19	65	12	42	3	3

Table 8.2: Usage of types in methods (*mth*) and classes (*cls*).

Table 8.2 presents the usage of some kinds of types in the projects of the corpus. These kinds of types are parametric, union, structural and combinations of structural and nominal types. We decide to include only these because they are not frequently used. Others types, such as nominal and self types, are core features of Gradualtalk and hence they are used pervasively. Additionally, the presence of these types in the corpus confirms that our decision to include them was correct. This is the case of parametric, union and structural types that have a significant number of usages, especially parametric types. In the case of union and structural types, we can argue that their verbosity prevents their wide use. On the other hand, combinations of structural and nominal types are very few. A possible reason for this is that such types are more complex to use compared to others, because of its subtyping notions (see Section 7.7.3). As a final remark, these results show the importance of parametric, union and structural types in Gradualtalk.

8.3 Bugs and refactoring

Bugs found. When typing the corpus, we found three bugs in the Kernel using Gradualtalk. The three bugs are present in Pharo Smalltalk 1.4 version #14438, and thanks to the feedback provided by our typing effort have since been fixed. Although this is a small quantity of bugs, the code being typed is very mature and hence can be expected to have a low number of bugs. That we still encountered bugs illustrates the advantages of typing code using Gradualtalk.

The first bug was found in the method silentlyValue in BlockClosure. The bug consists in the call to a method that does not exist, leading to an error at runtime. The bug was introduced because a method in Object was recently removed. However, there are classes in the system which still implement this method, making it difficult to statically detect this error without a typechecker.

The second bug was found in the method putOn: in Magnitude. With this bug, certain types of objects cannot be sent on a binary stream. The problem is that this method requires all subclasses of class Magnitude to implement the method asByteArray, while this is the case only for one subclass. The structural type {asByteArray (→ ByteArray)} was necessary to properly type this method.

The last bug was found in the method organization in ClassDescription. The problem is that it tries to invoke a recovery method when it finds a certain type of exception. However, that recovery method does not exist anywhere.

Refactoring. One of the main goals of Gradualtalk is to adapt itself to Smalltalk programming idioms. The typechecker does not require changes to the source code for typing, and many idiomatic uses of Smalltalk can be statically typed. Yet this still means that the programmer may need to resort to the dynamic type in some cases.

To make an existing code base more suitable for defining static types, a number of additional changes may be performed, ranging from small changes up to a complete redesign. Here we present three simple, optional changes to increase the amount of code that can be straightforwardly statically typed.

The first change consists of always adding a return to methods that raise an exception. The following code snippet shows an example of this case:

```
ClassDescription ≫ definition
   self subclassResponsibility
```

Because the method subclassResponsibility never returns normally, it does not matter what the method does after the invocation. However, following the Smalltalk semantics, this method returns self for the type checker. This is because it does not have any information to know that the last statement never returns. The solution is to make the code explicitly return self subclassResponsability and annotate the return type of this method to be the expected return type of the concrete implementations in subclasses. This typechecks because the return type of self subclassResponsability is the bottom type. This small refactoring allows the type system to properly check implementations of method definition in subclasses of ClassDescription. In contrast, just leaving the method untyped, *i.e.* returns Dyn, will not provide any type information, and this may lead to type inconsistencies in subclasses.

A second related change is adding abstract methods to classes when there is an implicit common selector between subclasses. This change is recommended because it indicates which methods need to be implemented in subclasses. However, if the developer does not want to implement this abstract method, she can use a Union type.

The third change is not to use "#()" to instantiate empty ordered collections, as this instantiates an Array object instead. Although this expression is shorter to write, and quite common throughout the code we typed, array has one important difference with ordered collections: its size cannot be changed. This already raises issues in Smalltalk, since when the developer tries to add an element to this object it throws an exception. The result is that there are potential errors hidden throughout the code. Currently programmers can deal with it by refactoring those usages to "#() asOrderedCollection" to obtain a guaranteed expandable collection. We believe that enforcing this change of idiom would make these guarantees more explicit and make typing these values easier. The latter is true because it would no longer require the use of a Union type of OrderedCollection and Array.

8.4 Interesting illustrations of Gradualtalk

We now describe a couple of methods from the corpus that showcase the combination of features of Gradualtalk.

Object ≫ #at:modify:

```
(a) at: (Integer)index modify: (Dyn→a) aBlock
    "Replace the element of the collection with itself transformed by the block"
    ↑ self at: index put: (aBlock value: (self at: index))
```

This method shows the usefulness of the Dyn type. The parameter aBlock is a closure that receives as a parameter the i-th element in the collection and returns the new element to be stored instead. However, the type of the originally stored element is unknown in Object. Declaring aBlock as (Object→a) would require the use of casts every time this method is used. This is the reason aBlock is typed as (Dyn→a).

Object ≫ #caseOf:otherwise:

```
(a|b) caseOf: (Collection<Association<→Object, →a>>) aBlockAssociationCollection
      otherwise: (→b) aBlock
    "The elements of aBlockAssociationCollection are associations between blocks.
     Answer the evaluated value of the first association in aBlockAssociationCollection
     whose evaluated key equals the receiver.  If no match is found, answer the result
     of evaluating aBlock."
    aBlockAssociationCollection associationsDo:
        [:(Association<→Object, →a>)assoc | (assoc key value = self) ifTrue: [↑assoc value
      value]].
    ↑ aBlock value
```

This method is the Smalltalk version of the switch statement of Java or C++. This interesting method shows various features of Gradualtalk being used, like type parameters (a), union types (a|b), function types for blocks (→a) and generic types (Association<→Object, →a>).

8.5 Typing challenges

When typing the corpus, we found some challenging methods to type using the current features of Gradualtalk. We now discuss some of them.

BlockClosure ≫ #whileFalse:

```
(Nil) whileFalse: (→Object) aBlock
    ...
```

This is one of the basic control methods in Smalltalk[1]. This method is problematic for Gradualtalk because it has conditions for its invocation, namely that the receiver must be a block of type (→Boolean) to be valid. In any other kind of block, invoking this method raises an exception. However, in Gradualtalk we cannot declare that a method in a given class can only be invoked on a subset of its instances. One possibility is to support a form of typestate checking [Wolff et al., 2011] but there is not enough evidence so far that such a feature would be sufficiently useful to warrant the added complexity.

Number ≫ #+

```
(Dyn) + (Number) aNumber
    "Refer to the comment in Number + "
    aNumber isInteger ifTrue:
        [self negative == aNumber negative
            ifTrue: [↑ (self digitAdd: (<Integer>aNumber) normalize)]
            ifFalse: [↑ self digitSubtract: (<Integer>aNumber)]].
    ↑ aNumber adaptToInteger: self andSend: #+
```

The method + is particularly challenging to type. The ideal type for this method would be one that could represent the type relation between receiver, argument and return shown in Table 8.3.

	Float	Fraction	SmallInteger	LargeInteger
Float	Float	Float	Float	Float
Fraction	Float	Fraction|Integer[1]	Fraction	Fraction
SmallInteger	Float	Fraction	Integer[2]	Integer[2]
LargeInteger	Float	Fraction	Integer[2]	Integer[2]

Table 8.3: Type relation for `Number >>> #+`. Rows correspond to the receiver type, columns correspond to the argument type and each cell value is the corresponding return type.

[1] Fractions are automatically simplified to Integer if applicable.
[2] The size of the integer is not guaranteed to be maintained.

[1]The curious reader may wonder why the method returns Nil instead of Self. The reason is that it is a special method that is inlined at call sites, where self is not bound to the block object.

However, this ideal is not expressible in Gradualtalk. Consider the case where the receiver is an Integer. We could type Integer ≫ + with four different types: Number→Dyn, Integer→Integer, Number→Number or (a <: Number)→a. The first type works, but it loses type information. The second type would reject the expression 2 + 3.5, requiring the manual coercion of 2 to a Float. The third type would require adding an explicit cast whenever the program does an arithmetic operation on Integers and stores it in a variable of type Integer (something which, not surprisingly, happens very often). Typing Integer≫+ as (a <: Number)→a is not correct either. Consider:

```
|(SmallInteger)x (LargeInteger)y|
y := (2 raisedTo: 10000).
x := y + 0.
```

In the expression y+0, y is the receiver, and the argument, 0, is a SmallInteger. So the type of y+0 would be SmallInteger, whereas numerically it clearly is not.

Number ≫ #to:by:do:

```
(Nil) to: (Number) stop by: (Number) step do: (Dyn→Object) aBlock
    "Normally compiled in–line, and therefore not overridable.
    Evaluate aBlock for each element of the interval (self to: stop by:
step)."
    | (Number)nextValue |
    nextValue := self.
    step = 0 ifTrue: [self error: 'step must be non–zero'].
    step < 0
        ifTrue: [[stop <= nextValue]
                whileTrue:
                    [aBlock value: nextValue.
                    nextValue := nextValue + step]]
        ifFalse: [[stop >= nextValue]
                whileTrue:
                    [aBlock value: nextValue.
                    nextValue := nextValue + step]].
        ↑nil.
```

This method is the Smalltalk "for" statement with steps. This method is problematic to type because the method Number ≫ #+ is difficult to type. The ideal type of this method is:

Number   (a <: Number)   ((Self|x) → Object)   → Nil

with x being the type of (self+step). Typing aBlock as (Number→Object) would force the Smalltalk programmer to either use only (Number→Object) or (Object →Object) closures, or add a cast to (Number→Object) in the closure. Our actual typing of aBlock (Dyn→Object) does not enforce correctness in this argument in the usage of the method, but does preserve usability.

Object ≫ #as:

```
(Dyn) as: (Object class) aSimilarClass
    "Create an object of class aSimilarClass that has similar contents to the receiver."
    ↑ aSimilarClass newFrom: self
```

This method creates a new object using the provided class and based on the contents of the receiver. This method is problematic to type in Gradualtalk, because the return type depends directly on the argument. However, this dependency relationship is instantiation. The ideal type of this method is (a→x), where x is the type of an instance of a. Typing this method as (Object class→Object), results in the programmer being required to add a cast to this expression, even if it would be obvious, making the code more verbose. The following code snippet is an example of this "obvious" cast:

```
|(ColorPoint)cp (Point)p|
cp := (<ColorPoint> p as: ColorPoint).
```

Currently, this method is typed as (Object class → Dyn), which removes the need for explicit casts, but it does not prevent misuse of the return value. A proper precise typing would be (a → a instance), but this is not (yet) supported in Gradualtalk.

BlockClosure >>> #on:do:

```
(Dyn) on: (Dyn) exception do: (BlockClosure)handlerAction
    "Evaluate the receiver in the scope of an exception handler."

    |(Boolean) handlerActive |
    <primitive: 199>  "just a marker, fail and execute the following"
    handlerActive := true.
    ↑ self value
```

This method is the exception handler of Smalltalk. This method is exceptionally challenging to type for three reasons:

1. This method is “magical”. The work of handling an exception and evaluating the closure handlerAction when the exception corresponds to the declared set of exceptions to catch, is done by the method ContextPart ≫ #handleSignal: using reflection on the call stack.

2. Like the method BlockClosure ≫ #whileFalse:, this method is only valid when invoked in zero parameter blocks. If a non-zero parameter block invokes it, it will raise an error. This error can be caught by the exception handler. For example, evaluating the expression ([:x|x] on: Error do: [42]) would return 42 instead of raising an error.

3. Typing this method is extremely difficult without losing some type information. A possible typing of this method is:

```
(Z|a) on: (X) exception do: (→a | Y→a) handlerAction
```

   with X being (Exception class|ExceptionSet), Y being the instance type of X, Z the type of the return value of the receiver, and a being a variable type (bounded to Object by default) representing the return value of handlerAction. Typing Y, without taking into account ExceptionSet, has the same problem as when typing the method Object ≫ #as:. Including ExceptionSet into the equation, would also require the support of heterogeneous collections.

8.6 Threats to validity

As in any study, this empirical validation of Gradualtalk suffers from a series of threats to validity.

For instance, we only typed seven open source projects (and in some cases, just partially), hence we cannot generalize that Gradualtalk will have the same level of usefulness in other Smalltalk projects as in our study. However, our corpus contains system projects as well as application projects, hence we expect that Gradualtalk will be useful in the vast majority of Smalltalk programs. In the case of partially typed projects, their overall results can change once they are fully typed. These changes can increase the percentage of Dyn if more dependencies to untyped code

are discovered. On the other hand, as more code is typed, programmers receive more benefits, *e.g.* more bugs are detected.

We find that most of the usage of the dynamic type in the typed projects is due to dependencies to untyped code. These dependencies can be to the same project (*i.e.* the part that is not yet typed) or external projects. Hence, depending of which part of the project is typed or its dependencies to external projects, the percentage of Dyn and the results in Table 8.1 may vary. However, as we mentioned, these percentages reflect different stages of migrating untyped to typed code. As more code is typed, we can expect less dynamic dependencies. Consequently, as we continue migrating more Smalltalk projects to Gradualtalk, the dependencies on untyped code will gradually decrease.

Though we carefully and systematically decide what type will be the best fit for each variable, method or block, there exists the possibility that these types may be biased to the programer's self understanding of these projects. This can affect the kinds of type used, especially complex ones, however we (*i.e.* the four people included in the study) discuss the typing of complex cases.

We introduce types in a methodology that may favor nominal and self types instead of more complex ones, *e.g.* parametric, union and structural types. This may affect the overall results on Table 8.2. However a complex type does not add more type information than a simple one, if the latter can replace the former. Hence a higher or lower number of complex types does not negatively affect the usefulness of Gradualtalk.

8.7 Conclusions

We typed seven Smalltalk projects and reported our experience as an early validation of Gradualtalk. We find that migrating untyped code to their typed versions is a continuous task that requires significant work. At the beginning, you might expect a high use of Dyn, though they will decrease gradually as typed modules increase. This is mainly because of the dependencies of typed modules to untyped modules. Table 8.1 shows the different stages of this migration.

We also report our findings regarding typing features, code refactoring, bug fixes, and limitations of Gradualtalk:

- Table 8.2 shows the relevance of some kinds of types (*i.e.* parametric, union and structural types) that are not commonly used by programmers. We find that parametric types are significantly used in our corpus. This confirms our decision to include generics regardless of its controversies. Additionally we find that union and structural types are fairly used across the corpus. On the other hand, combinations of structural and nominal types are rarely used.

- Refactoring is usually not considered as an alternative for retrofitted type systems. Additionally in Gradualtalk, refactoring is not required, because complex expressions can be left untyped. However, we show that small code refactoring can significantly improve the type system to provide better type information, and hence better feedback for programmers.

- We report on three bugs that were found by Gradualtalk while typing the corpus. Although this number of bugs is small, the projects in the corpus are mature Smalltalk software that have a meticulous testing process. This shows the usefulness of Gradualtalk even on highly tested code.

- Finally, we report on typing challenging methods that showcase the limitations of Gradualtalk and the particularities and oddities of Smalltalk. In these cases, the use of `Dyn` allows us to type them. However the consequence of using `Dyn` is the loss of type information.

These findings show the relevance of Gradualtalk for Smalltalk developers. Furthermore, these results show that our design decisions were correct (at least for the first version). While we will continue to add types to more existing libraries ourselves, the most interesting feedback on the usefulness of Gradualtalk will come from the user community, which will allow us to refine the selection of type system features and deepen our understanding of how certain features are used (or not).

The most pressing challenge to ensure the wide adoption of Gradualtalk is performance. Based on our experience so far, gradually-typed applications run significantly slower (two orders of magnitude) than their dynamically-typed version. While certain techniques to optimize gradual typing have been proposed [Herman et al., 2010; Siek and Wadler, 2010], it is unclear yet which techniques are most

effective in the specific context of Smalltalk, and if it is necessary to devise new optimization strategies for this context.

Part III: Conclusions

Chapter 9

Contributions

As dynamically typed languages become more relevant in large software development, they suffer from problems that type systems can easily solve. However the introduction of types in those languages is complex and, without a proper guide, type system designers may end up developing a retrofitted type system that does not fit the dynamically typed language. In this dissertation, we argued that empirical evidence improves the design and implementation of pragmatic retrofitted type systems. To validate our thesis, we empirically designed and implemented Gradualtalk, a retrofitted type system for Smalltalk. We validated it by typing several Smalltalk projects.

This chapter briefly reviews the main contributions of this thesis work. Our contribution can be divided in two different topics:

- **Gradualtalk**. A retrofitted type system for Smalltalk (chapters 3, 7 and 8).
- **Empirical studies**. Two large-scale and three preliminary empirical studies on a large Smalltalk code base (chapters 4, 5 and 6).

The following sections summarize these contributions.

9.1 Gradualtalk

Our first main contribution is the development of a retrofitted type system for Smalltalk. This is a gradual type system carefully extended to cover Smalltalk features and idioms as much as possible. Developing such a type system requires meeting several achievements: a type system design that properly fits Smalltalk without unduly increasing its complexity; implementing it in a live environment; and a validation of the system by typing a nontrivial corpus of code.

We proposed a novel empirically-driven design of Gradualtalk which can be divided into three steps. We first carefully analyze Smalltalk to find its relevant features and idioms from a type system perspective. We then explicitly define the type system goals based on three aspects: errors to avoid, guarantees and flexibility. These goals are the guidelines to choose gradual typing as the most suitable partial type system for Smalltalk. Finally, we propose a set of typing extensions targeted to cover Smalltalk features and idioms. Each typing extension is properly discussed and justified with empirical evidence (from the literature and our presented empirical studies) that assisted us to make informed decisions on them.

The results of the design is a practical gradual type system for Smalltalk that supports a smooth path from untyped to typed code. In fact, we show that any Smalltalk program is a valid Gradualtalk program, and type annotations can be added selectively per expressions. Gradualtalk presents a novel combination of typing features, with some interesting interactions. We also briefly discuss

Gradualtalk safety and type soundness, and the challenges to implement it in a live system.

Finally, we validate Gradualtalk through typing seven Smalltalk projects. Those projects are a mixed selection of system libraries and applications. In total, we typed 18,780 LOC, 3,382 methods in 137 classes in this corpus. We report on found bugs, refactored code and challenging methods to type.

We believe that Gradualtalk would have a significant impact in the Smalltalk community in the long-term. This is because of its advantages, *e.g.* early error detection, that it introduces for developing software. Currently, this first version of the type system requires more work to be really useful in an industrial setting. Some important issues (as mentioned throughout this document) are performance and the lack of fully typed APIs. The former is currently under research [Allende et al., 2014b, 2013], and the latter requires a deeper adoption of Gradualtalk in the community. Nevertheless this work presents a novel methodology for designing pragmatic retrofitted type systems that can be applied to introduce types in another (perhaps more widely used) dynamically-type language, such as JavaScript.

9.2 Empirical studies

In this thesis work, we presented two large-scale empirical studies over 1,000 Smalltalk projects extracted from the SqueakSource repository. In addition, we performed three preliminary empirical studies over a set of 139 Smalltalk projects from Pharo base image (version 1.2.1), web framework Seaside and related subprojects. The guidelines provided by those studies help us make a more practical implementation of Gradualtalk. Additionally, those studies are contributions per se that can be useful for other researchers and in other areas. In the following paragraphs, we summarize the contributions of each empirical study.

The dynamic and reflective features of Smalltalk.

Chapter 5 explores the usage of dynamic features in Smalltalk in order to gain insight on the usage of these features in practice. In this study, we performed a quantitative and qualitative analysis of a large corpus of Smalltalk code base (more than 4 million LOC). In the quantitative analysis, we research how (and how often)

developers use the dynamic features of Smalltalk. We find that dynamic features are not used often, and actually those used are more recurrent in specific kinds of software, like Tests or System, than in general applications. Additionally, we performed a qualitative analysis on a representative sample of 377 dynamic feature usages in order to better understand the reasons why developers resort to using these dynamic features. We find that a large portion of usages are genuine. These results offer us insight into how language designers and tool providers have to deal with these dynamic features, and into why the developers need to use them.

In the particular case of Gradualtalk, this study allows us to make informed decisions on the typing features: self types, incremental type checking and effect systems. In the case of self types, this study shows the importance of self to instantiate objects and create classes. For incremental type checking and effect systems, this study shows that reflective features are rarely used. Therefore incremental type checking is more suitable to be implemented in Gradualtalk than an effect system. Additionally, this study helps us understand the idiom "symbols as methods", whose usages may be mostly unsafe, and hence hardly tractable statically. These guidelines help us make informed decisions on the design of Gradualtalk.

Type predicates and type-based dispatch patterns in Smalltalk.
In this second empirical work we study if, and how, type predicates are used in practice. Specifically, we answered four research questions about the prevalence of type predicates and logical combinations of them, kind of type predicates usages, and if those predicates are constant at runtime. We find that programmers use a fair number of type predicates for explicit dispatch, and consequently this idiom is a practical problem. However more than three quarters of those usages only check for object nullness—a specific case. Additionally, we find that a good portion of predicates may be constant over time, though our results are not conclusive. As a final remark, we concluded that flow-sensitive typing approaches are useful for objects.

This study helps us in Gradualtalk to decide on deferring flow-sensitive typing to a next version. This is because the prevalence of type predicates to do explicit dispatch is too dispersed (one for each 50 lines of code) to be considered a main

feature in the first version of Gradualtalk. However, we are planning to include this feature in an upcoming version.

Preliminary empirical studies

These are a series of empirical studies over the Pharo 1.2.1 base image, the Seaside web framework and other minor projects. We study the use of self as a return value, the use of variables that may represent different types, and the use of collections. On the use of self, we find that almost half of the methods return self, showing the importance of self types. On the use of variables that join several types, we find that 1,035 methods (1.4% of the corpus) use variables that may represent several types at the same time. This result plus the low effort to introduce union types helps us decide on extending Gradualtalk with union types. Finally, on the use of collections, we find that 3 out of 10 documented usages of collections specify the kind of collection, and almost 98% of collections at run time have elements of the same class (different than Object) or its subclasses. These results suggest that generics are relevant for Gradualtalk. Although the results of these studies are limited or specific to some scenarios, they show tendencies that help us make informed decisions in the type system, and possible new research paths for researchers.

Chapter 10

Perspectives

The contributions of this thesis work directly opens new perspectives for future work. Based on its limitations and some ideas originating from its development, we propose a number of potential directions for future research. This chapter briefly discusses the most important ones.

The structure of this chapter follows the structure of the previous chapter: Perspectives regarding Gradualtalk are first discussed in Section 10.1. Section 10.2 summarizes perspectives with respect to future focus on empirical studies based on the results presented in this thesis work and open questions that we identified.

10.1 Gradualtalk

Gradualtalk 2.0 and controlled experiments

The implemented version of Gradualtalk presented in this work is just the first iteration of a development cycle of a retrofitted type system for Smalltalk. Gradualtalk presents state-of-the-art typing features that aid programmers in properly typing Smalltalk code. However, as we show in Chapter 8, some features are not used a lot, and others are still missing. In this regard, we are still evaluating the effectiveness of Gradualtalk to better understand its real contribution to programmers. We expect to receive feedback from the Smalltalk community to update Gradualtalk to their needs. Such updates may include more typing features, or remove others that showed to be rarely used or unhelpful. Additionally, the real applicability of

Gradualtalk is not yet measured or proved to be effective. Therefore, a series of controlled experiments can help us with this issue.

A soundness proof for Gradualtalk

This thesis work is experimental rather than theoretical, proofs of that are the empirical studies and the development of the type system. Although, we have presented in detail the notions of the type system semantics (*i.e.* some interactions between other kinds of types, subtyping, and soundness concept) in Section 7.9, a proper soundness proof for Gradualtalk is still missing. The lack of such formal proof reduces the impact of Gradualtalk, because we cannot claim strong guarantees for Gradualtalk, nor we can be sure if there are problems, such as undecidability, due to the combinations of typing features. Therefore, a soundness proof for Gradualtalk is the natural next step in this research.

Tools support

The type information provided by Gradualtalk opens new possibilities for research in developer tools. For instance, IDE tools can use such information to improve code completion, syntax highlighting and source code navigation. Another opportunity is in source code versioning systems, such as Monticello, where type annotations are currently not properly supported. The use of type information can simplify the binding process between classes and methods in Monticello. As a third example, debuggers can use type information, such as bad cast reports, to better pointing the case of exceptions. Gradualtalk opens new possibilities of research and enhancements in developer tools for Smalltalk.

10.2 Empirical studies

Dynamic features in other languages

Chapter 5 presents an empirical study on the dynamic features of Smalltalk. However, we cannot generalize our findings to other mainstream languages, such as Ruby. Our empirical study was specifically targeted to Smalltalk to help us to make informed decisions about typing extensions on Gradualtalk. This is also the case of studies in Python [Åkerblom et al., 2014] and Javascript [Richards et al.,

2010]. However, an empirical study in another mainstream dynamically typed language would give more confidence or refute the conclusions in Chapter 5. New results can help practitioners and researchers to better understanding the use of dynamic features in dynamically typed languages.

Are dynamic features executed?

Chapter 5 presents a static analysis on the prevalence of dynamic features in Smalltalk. Although this is a large scale empirical study (more than one thousand Smalltalk projects), it is limited to a static analysis, hence we do not know if the dynamic features are executed or not. This limitation reduces the impact of the results, but performing a dynamic analysis in such large corpus is hard to achieve and out-of-scope of this thesis work. Determining whether or not dynamic features are executed would provide new information that may change their relevance in the type system. Therefore, this a new research path that we must explore to improve Gradualtalk.

Type predicates impact on refactoring and teaching

Chapter 5 presents an empirical study on the prevalence and usages of type predicates in Smalltalk. We use those results to make informed decisions for designing Gradualtalk. However, those results and findings can also contribute to the discussion of type predicates in other areas, *e.g.* refactoring and teaching. In refactoring, those results may assist practitioners when they attempt to remove usages of such predicates. In education, current pedagogical approaches would benefit from contrasting the core principle of the object-oriented paradigm with the state-of-the-practice, raising awareness about the typical pitfalls and design alternatives. Hence, the results in chapter 5 open new research paths to understanding object oriented software in practice.

State predicates

The framework that we developed to trace type predicates in Smalltalk can be easily adapted to trace state predicates. State predicates are those methods isXxxx that return a boolean based on the state of an object. A possible line of future

research is to apply modified version of the static and dynamic analyzers of chapter 6 to investigate the prevalence and usages of state-based predicates. This would inform researchers and practitioners about the relevance of complex state-tracking typing techniques, like typestate.

Joining different values

Programmers in dynamically typed languages frequently allow variables to refer to values of different types. For example, consider the following method header (extracted from Pharo 3): `Socket>> receiveDataInto: aStringOrByteArray`. This method receives a data into the given buffer (`aStringOrByteArray`) that may be either a string or a byte array. Other scenarios are those where methods may return different values on different cases, *e.g.* returning `false` or an element in a look up method. In fact, in a preliminary empirical study (see Section 4.4), we find that these kinds of idioms are potentially present. However, a large-scale and deeper empirical study is required to properly understand its relevance, and consequently make stronger claims. Studying this pattern would inform type system designers and practitioners on the relevance of union types on retrofitted type systems. Furthermore, such a study may answer such questions as: Can those cases be replaced with structural types or a common ancestor? Do those variables use type predicates to disambiguate their type?

Are symbols a replacement for methods?

In Smalltalk, we find pieces of code like `array do: #printString` instead of `array do: [:e| e printString]`. This particular idiom can be hard to cover by a type system, because it uses reflection and some scenarios are undecidable. However, a couple of questions arise: How prevalent is this idiom? Are there some variants? If there are several variants of this idiom, how many can be statically tractable? Actually, refactoring those usages to a more tractable one may mitigate this issue, but is it a feasible option? Answering these questions would inform type system designers and practitioners in making informed decisions regarding the prevalence and tractability of this idiom by static analyzer tools.

Unit tests in practice

Unit testing is a common and proven method to ensure that software behaves as intended. Although the benefits of testing are clear, we often find software that is partially or entirely untested. Proof of this is that just 562 out of 1,000 Smalltalk projects in our corpus include test cases. Actually, we use unit tests as scenarios for performing the dynamic analysis in Section 6.6.5, but we cannot claim that those scenarios are representative benchmarks of the measured software. Hence, another path for research is to investigate the prevalence of unit tests and their relevance. This would help practitioners to make informed decisions on how to test their software, and provide researchers with opportunities to propose better tools and methodologies for testing software.

Bibliography

Abadi, M., Cardelli, L., Pierce, B., and Plotkin, G. (1991). Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268. 10

Åkerblom, B., Stendahl, J., Tumlin, M., and Wrigstad, T. (2014). Tracing dynamic features in Python programs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 292–295. 114, 191

Aldrich, J., Sunshine, J., Saini, D., and Sparks, Z. (2009). Typestate-oriented programming. In *Proceedings of Onward!*, pages 1015–1022. ACM. 106, 147

Allende, E., Callaú, O., Fabry, J., Tanter, É., and Denker, M. (2014a). Gradual typing for Smalltalk. *Science of Computer Programming*, 96(1):52–69. 6

Allende, E., Fabry, J., Garcia, R., and Tanter, É. (2014b). Confined gradual typing. In *Proceedings of the 29th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2014)*, Portland, OR, USA. ACM Press. To appear. 187

Allende, E., Fabry, J., and Tanter, É. (2013). Cast insertion strategies for gradually-typed objects. In *Proceedings of the 9th ACM Dynamic Languages Symposium (DLS 2013)*, pages 27–36, Indianapolis, IN, USA. ACM Press. ACM SIGPLAN Notices, 49(2). 16, 187

Ancona, D., Ancona, M., Cuni, A., and Matsakis, N. D. (2007). RPython: a step towards reconciling dynamically and statically typed oo languages. In *Proceedings of the 2007 symposium on Dynamic languages*, DLS '07, pages 53–64. 19

Anderson, C., Drossopoulou, S., and Giannini, P. (2005). Towards type inference for javascript. In Black, A. P., editor, *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, number 3586 in Lecture Notes in Computer Science, pages 428–452, Glasgow, UK. Springer-Verlag. 19

Andreae, C., Noble, J., Markstrum, S., and Millstein, T. (2006). A framework for implementing pluggable type systems. *SIGPLAN Not.*, 41(10):57–74. 11

Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 2

Beckman, N. E., Kim, D., and Aldrich, J. (2011). An empirical study of object protocols in the wild. In Mezini [2011], pages 2–26. 147

Bergel, A., Cassou, D., Ducasse, S., and Laval, J. (2013). *Deep into Pharo*. Square Brackets Associates. 27

Bierhoff, K. and Aldrich, J. (2007). Modular typestate checking of aliased objects. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2007)*, pages 301–320, Montreal, Canada. ACM Press. ACM SIGPLAN Notices, 42(10). 147

Black, A., Ducasse, S., Nierstrasz, O., Pollet, D., Cassou, D., and Denker, M. (2007). *Squeak by Example*. Square Bracket Associates. http://SqueakByExample.org/. 11

Bloch, J. (2008). *Effective Java, 2nd Edition*. Addison-Wesley. 120, 124

Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., and Mezini, M. (2011). Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 241–250. 65, 85, 114

Boyland, J. T. (2014). The problem of structural type tests in a gradual-typed language. In *21th International Workshop on Foundations of Object-Oriented Languages*, FOOL 2014. 149

Bracha, G. (1997). The strongtalk type system for Smalltalk. http://www.bracha.org/nwst.html. 19

Bracha, G. (2004). Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, pages 1–6. 11, 20, 25, 35

Bracha, G. (2011). Optional types in dart. https://www.dartlang.org/articles/optional-types/. 12

Bracha, G. and Cook, W. (1990). Mixin-based inheritance. In Meyrowitz, N., editor, *Proceedings of the 5th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA/ECOOP 90)*, pages 303–311, Ottawa, Canada. ACM Press. ACM SIGPLAN Notices, 25(10). 29

Bracha, G. and Griswold, D. (1993). Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings of the 8th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 95)*, pages 215–230, Washington, D.C., USA. ACM Press. ACM SIGPLAN Notices, 28(10). 19, 20, 25, 71, 93, 100, 153

Callaú, O., Robbes, R., Tanter, É., and Röthlisberger, D. (2013). How (and why) developers use the dynamic features of programming languages: the case of Smalltalk. *Empirical Software Engineering*, 18(6):1156–1194. 6

Callaú, O., Robbes, R., Tanter, É., Röthlisberger, D., and Bergel, A. (2014). On the use of type predicates in object-oriented software: The case of Smalltalk. In *Proceedings of the 10th ACM Dynamic Languages Symposium (DLS 2014)*, pages 135–146, Portland, OR, USA. ACM Press. 6

Callaú, O. and Tanter, É. (2013). Programming with ghosts. *IEEE Software*, 30(1):74–80. 6, 163, 170

Cardelli, L. (1997a). Type systems. In Tucker, A. B., editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press. 1, 9

Cardelli, L. (1997b). Type systems. In Tucker, A. B., editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press. 167

Cartwright, R. and Fagan, M. (1991). Soft typing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 278–292. 11, 35

Castagna, G., editor (2009). *Proceedings of the 18th European Symposium on Programming Languages and Systems (ESOP 2009)*, volume 5502 of *Lecture Notes in Computer Science*, York, UK. Springer-Verlag. 202, 208

Chalin, P. and James, P. R. (2007). Non-null references by default in Java: Alleviating the nullity annotation burden. In *ECOOP 2007: Proceedings of the 21st European Conference on Object-Oriented Programming*, pages 227–247. 145

Chang, M., Mathiske, B., Smith, E., Chaudhuri, A., Gal, A., Bebenita, M., Wimmer, C., and Franz, M. (2007). The impact of optional type information on JIT compilation of dynamically-typed languages. In *Proceedings of the ACM Dynamic Languages Symposium (DLS 2007)*, pages 13–24, Montreal, Canada. ACM Press. 11

Cook, W. R. (2009). On understanding data abstraction, revisited. *ACM SIGPLAN Notices*, 44(10):557–572. 120

de Lausanne (EPFL), É. P. F. (2014). Scala. http://www.scala-lang.org. 161

DeLine, R. and Fähndrich, M. (2004). Typestates for objects. In Odersky, M., editor, *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, number 3086 in Lecture Notes in Computer Science, pages 465–490, Oslo, Norway. Springer-Verlag. 121, 136, 147

Dietl, W., Dietzel, S., Ernst, M. D., Muşlu, K., and Schiller, T. W. (2011). Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 681–690. 11

Doligez, D., Frisch, A., Garrigue, J., Rémy, D., and Vouillon, J. (2011). *The OCaml system release 3.12*. Institut National de Recherche en Informatique et en Automatique, http://caml.inria.fr/pub/docs/manual-ocaml/index.html. 160

Ekman, T. and Hedin, G. (2007). Pluggable checking and inferencing of non-null types for Java. *TOOLS EUROPE 2007, Journal of Object Technology*. 11

Erdös, K. and Sneed, H. M. (1998). Partial comprehension of complex programs (enough to perform maintenance). In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, page 98, Washington, DC, USA. IEEE Computer Society. 96

Facebook (2014). The hack language. http://hacklang.org. 11

Fagan, M. (1990). *Soft Typing: An Approach to Type Checking for Dynamically Typed Languages*. PhD thesis, Rice University. 11

Fähndrich, M. and Leino, K. R. M. (2003). Declaring and checking non-null types in an object-oriented language. In Crocker, R. and Steele, Jr., G. L., editors, *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003)*, pages 302–312, Anaheim, CA, USA. ACM Press. ACM SIGPLAN Notices, 38(11). 121, 124

Figueroa, I., Tanter, É., and Tabareau, N. (2012). A practical monadic aspect weaver. In *Proceedings of the 11th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2012)*, pages 21–26, Potsdam, Germany. ACM Press. 17

Flanagan, C. (2006). Hybrid type checking. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 245–256, New York, NY, USA. ACM. 13

Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional. 118

Freeman, T. and Pfenning, F. (1991). Refinement types for ML. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 268–277, New York, NY, USA. ACM. 13

Furr, M. (2009). *Combining Static and Dynamic Typing in Ruby*. PhD thesis, University of Maryland. 12, 17

Garcia, R., Wolff, R., Tanter, É., and Aldrich, J. (2010). Featherweight typestate. Technical Report CMU-ISR-10-115, Carnegie Mellon University. 35

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and its Implementation*. Addison-Wesley. 25, 65

Graver, J. O. and Johnson, R. E. (1990). A type system for Smalltalk. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 136–150. 20

Grechanik, M., McMillan, C., DeFerrari, L., Comi, M., Crespi, S., Poshyvanyk, D., Fu, C., Xie, Q., and Ghezzi, C. (2010). An empirical investigation into a large-scale Java open source code repository. In *ESEM '10: Proceedings of the 4th International Symposium on Empirical Software Engineering and Measurement*, pages 11:1–11:10. 113

Gronski, J., Knowles, K., Tomb, A., Freund, S. N., and Flanagan, C. (2006). Sage: Hybrid checking for flexible specifications. In *In Workshop on Scheme and Functional Programming*. 13

Guha, A., Saftoiu, C., and Krishnamurthi, S. (2011). Typing local control and state using flow analysis. In Barthe, G., editor, *Proceedings of the 20th European Symposium on Programming (ESOP 2011)*, volume 6602 of *Lecture Notes in Computer Science*, pages 256–275. Springer-Verlag. 43, 118, 119, 121, 135, 145

Haldiman, N., Denker, M., and Nierstrasz, O. (2009). Practical, pluggable types for a dynamic language. *Comput. Lang. Syst. Struct.*, 35(1):48–62. 11, 20, 35

Herman, D., Tomb, A., and Flanagan, C. (2010). Space-efficient gradual typing. *Higher-Order and Sympolic Computation*, 23(2):167–189. 12, 182

Holkner, A. and Harland, J. (2009). Evaluating the dynamic behaviour of Python applications. In *ACSC '09: Proceedings of the 32nd Australasian Computer Science Conference*, pages 17–25. 114

Hoppe, M. and Hanenberg, S. (2013). Do developers benefit from generic types?: An empirical comparison of generic and raw types in Java. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 457–474. 45

Igarashi, A., Pierce, B. C., and Wadler, P. (2001). Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450. 33

Ina, L. and Igarashi, A. (2011). Gradual typing for generics. In *Proceedings of the 26th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2011)*, pages 609–624, Portland, Oregon, USA. ACM Press. 12, 155, 156, 167

Johnson, R. E. (1986). Type-checking Smalltalk. *SIGPLAN Not.*, 21(11):315–321. 20

Johnson, R. E., Graver, J. O., and Zurawski, L. W. (1988). Ts: an optimizing compiler for Smalltalk. *SIGPLAN Not.*, 23(11):18–26. 20

Knowles, K. and Flanagan, C. (2010). Hybrid type checking. *ACM Transactions on Programming Languages and Systems*, 32(2):Article n.6. 35

Knuth, D. E. (1971). An empirical study of FORTRAN programs. *Software: Practice and Experience*, 1(2):105–133. 112

Krishnamurthi, S. (2007). *Programming Languages: Application and Interpretation*. Version 2007-04-26. 9

Krishnamurthi, S., Felleisen, M., and Friedman, D. P. (1998). Synthesizing object-oriented and functional design to promote re-use. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECOOP '98, pages 91–113, London, UK, UK. Springer-Verlag. 125

Laforge, G. (2012). Whats new in Groovy 2.0? http://www.infoq.com/articles/new-groovy-20. 118

Lerner, B. S., Politz, J. G., Guha, A., and Krishnamurthi, S. (2013). Tejas: Retrofitting type systems for javascript. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 1–16. 12, 14, 18

Lungu, M., Robbes, R., and Lanza, M. (2010). Recovering inter-project dependencies in software ecosystems. In *ASE'10: Proceedings of the 25th IEEE/ACM international conference on Automated Software Engineering*, ASE '10, pages 309–312. 66, 68, 122, 139

Malayeri, D. and Aldrich, J. (2008). Integrating nominal and structural subtyping. In Vitek, J., editor, *Proceedings of the 22nd European Conference on Object-oriented Programming (ECOOP 2008)*, number 5142 in Lecture Notes in Computer Science, pages 260–284, Paphos, Cyprus. Springer-Verlag. 160, 161

Malayeri, D. and Aldrich, J. (2009). Is structural subtyping useful? an empirical study. In Castagna [2009], pages 95–111. 40, 113, 146, 160

Melton, H. and Tempero, E. D. (2007). An empirical study of cycles among classes in Java. *Empirical Software Engineering*, 12(4):389–415. 113

Meyers, S. (2005). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley. 118

Mezini, M., editor (2011). *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP 2011)*, volume 6813 of *Lecture Notes in Computer Science*, Lancaster, UK. Springer-Verlag. 196, 204, 208

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375. 2

Muschevici, R., Potanin, A., Tempero, E. D., and Noble, J. (2008). Multiple dispatch in practice. In *OOPSLA '08: Proceedings of the 23rd ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 563–582. 113

Nierstrasz, O., Ducasse, S., and Demeyer, S. (2009). *Object-Oriented Reengineering Patterns*. Square Bracket Associates. 118

Nierstrasz, O., Ducasse, S., and Pollet, D. (2010). *Pharo by Example*. Square Brackets Associates. 26, 27

Oliveira, B. C. (2009). Modular visitor components. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 269–293, Berlin, Heidelberg. Springer-Verlag. 125

Palsberg, J. and Schwartzbach, M. I. (1990). Type substitution for object-oriented programming. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, pages 151–160. 20

Palsberg, J. and Schwartzbach, M. I. (1991). Object-oriented type inference. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 146–161. 20

Papi, M. M., Ali, M., Correa, Jr., T. L., Perkins, J. H., and Ernst, M. D. (2008). Practical pluggable types for Java. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212, New York, NY, USA. ACM. 11

Parnin, C., Bird, C., and Murphy-Hill, E. (2013). Adoption and use of Java generics. *Empirical Software Engineering*, 18(6):1047–1089. 45, 113, 147

Pearce, D. (2013a). Sound and complete flow typing with unions, intersections and negations. In Giacobazzi, R., Berdine, J., and Mastroeni, I., editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7737 of *Lecture Notes in Computer Science*, pages 335–354. Springer Berlin Heidelberg. 118, 119, 121, 136, 146

Pearce, D. J. (2013b). A calculus for constraint-based flow typing. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs*, FTfJP '13, pages 7:1–7:7. 146

Pearce, D. J. and Noble, J. (2011). Implementing a language with flow-sensitive and structural typing on the JVM. *Electronic Notes in Theoretical Computer Science*, 279(1):47 – 59. Proceedings of the Bytecode 2011 workshop, the Sixth Workshop on Bytecode Semantics, Verification, Analysis and Transformation. 146

Pierce, B. C. (2002). *Types and programming languages*. MIT Press, Cambridge, MA, USA. 1, 8, 15, 35, 155, 157, 159, 160, 167

Pierce, B. C. (2005). *Advanced topics in types and programming languages*. MIT press. 39, 41

Pluquet, F., Marot, A., and Wuyts, R. (2009). Fast type reconstruction for dynamically typed programming languages. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 69–78. 20

Posnett, D., Filkov, V., and Devanbu, P. (2011). Ecological inference in empirical software engineering. In *Proceedings of the 26th ACM/IEEE International Conference on Automated Software Engineering (ASE 2011)*, pages 362–371. 129

Rastogi, A., Chaudhuri, A., and Hosmer, B. (2012). The ins and outs of gradual type inference. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2012)*, pages 481–494, Philadelphia, USA. ACM Press. 12, 19

Richards, G., Lebresne, S., Burg, B., and Vitek, J. (2010). An analysis of the dynamic behavior of Javascript programs. In *PLDI '10: Proceedings of the 31st ACM conference on Programming Language Design and Implementation*, pages 1–12. 114, 191

Richards, G., Lebresne, S., Burg, B., and Vitek, J. (2011). The eval that men do: A large-scale study of the use of eval in JavaScript applications. In Mezini [2011], pages 52–78. 101, 114

Robbes, R., Röthlisberger, D., and Tanter, É. (2012). Extensions during software evolution: Do objects meet their promise? In Noble, J., editor, *Proceedings of the 26th European Conference on Object-oriented Programming (ECOOP 2012)*, volume 7313 of *Lecture Notes in Computer Science*, pages 28–52, Beijing, China. Springer-Verlag. 125, 146

Roberts, D., Brant, J., and Johnson, R. (1997). A refactoring tool for Smalltalk. *Theor. Pract. Object Syst.*, 3(4):253–263. 139

Rysselberghe, F. V. and Demeyer, S. (2007). Studying versioning information to understand inheritance hierarchy changes. In *MSR '07: Proceedings of the 4th International Workshop on Mining Software Repositories*, page 16. 113

Saito, C. and Igarashi, A. (2009). Self type constructors. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 263–282, New York, NY, USA. ACM. 39, 40, 51, 153, 156, 167

Schärli, N., Black, A., and Ducasse, S. (2004). Object-oriented encapsulation for dynamically-typed languages. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2004)*, pages 130–149, Vancouver, British Columbia, Canada. ACM Press. ACM SIGPLAN Notices, 39(11). 105

Schärli, N., Ducasse, S., Nierstrasz, O., and Black, A. (2003). Traits: Composable units of behavior. In Cardelli, L., editor, *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP 2003)*, number 2743 in Lecture Notes in Computer Science, pages 248–274, Darmstadt, Germany. Springer-Verlag. 29, 160

Siek, J., Garcia, R., and Taha, W. (2009). Exploring the design space of higher-order casts. In *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 17–31, Berlin, Heidelberg. Springer-Verlag. 12

Siek, J. and Taha, W. (2006). Gradual typing for functional languages. In *Proceedings of the Scheme and Functional Programming Workshop*, pages 81–92. 10, 12, 25, 35, 168

Siek, J. and Taha, W. (2007). Gradual typing for objects. In Ernst, E., editor, *Proceedings of the 21st European Conference on Object-oriented Programming (ECOOP 2007)*, number 4609 in Lecture Notes in Computer Science, pages 2–27, Berlin, Germany. Springer-Verlag. 12, 160, 168, 169

Siek, J. and Wadler, P. (2010). Threesomes, with and without blame. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2010)*, pages 365–376, Madrid, Spain. ACM Press. 182

Siek, J. G. and Vachharajani, M. (2008). Gradual typing with unification-based inference. In *DLS '08: Proceedings of the 2008 symposium on Dynamic languages*, pages 1–12, New York, NY, USA. ACM. 12

Smit, R. (2012). Pegon. http://sourceforge.net/projects/pegon/. 20

Strom, R. E. and Yemini, S. (1986). Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171. 121, 136, 147

Takikawa, A., Strickland, T. S., Dimoulas, C., Tobin-Hochstadt, S., and Felleisen, M. (2012). Gradual typing for first-class classes. In *Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2012)*, pages 793–810, Tucson, AZ, USA. ACM Press. 12

Tempero, E. D. (2009). How fields are used in Java: An empirical study. In *ASWEC '09: Proceedings of the 20th Australian Software Engineering Conference*, pages 91–100. 113

Tempero, E. D., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. (2010). The qualitas corpus: A curated collection of Java code

for empirical studies. In *APSEC 2010: Proceedings of the 17th Asia Pacific Software Engineering Conference*, pages 336–345. 144

Tempero, E. D., Noble, J., and Melton, H. (2008). How do Java programs use inheritance? an empirical study of inheritance in Java software. In *ECOOP '08: Proceedings of the 22nd European Conference on Object-Oriented Programming*, pages 667–691. 113

Thies, A. and Bodden, E. (2012). Refaflex: Safer refactorings for reflective Java programs. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 1–11. 139

Tobin-Hochstadt, S. (2010). *Typed Scheme: From Scripts to Programs*. PhD thesis, Northeastern University. 11, 16, 118

Tobin-Hochstadt, S. and Felleisen, M. (2008). The design and implementation of Typed Scheme. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*, pages 395–406, San Francisco, CA, USA. ACM Press. 118, 133, 135, 145

Tobin-Hochstadt, S. and Felleisen, M. (2010). Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN Conference on Functional Programming (ICFP 2010)*, pages 117–128, Baltimore, Maryland, USA. ACM Press. 118, 119, 121, 133, 135, 145

Tobin-Hochstadt, S. and St-Amour, V. (2012). The typed racket guide. http://docs.racket-lang.org/ts-guide/. 16

Torgersen, M. (2004). The expression problem revisited four new solutions using generics. In *In Proceedings of the 18th European Conference on Object-Oriented Programming*, pages 123–143. Springer-Verlag. 125

Tratt, L. (2009). Dynamically typed languages. *Advances in Computers*, 77:149–184. 8, 9

Tratt, L. and Welc, A. (2014). An interview with Gilad Bracha. *Software, IEEE*, 31(5):76–79. 45

Triola, M. (2006). *Elementary Statistics*. Addison-Wesley, 10th edition. 95

van Rossum, G. (2004). Adding optional static typing to python. https://www.artima.com/weblogs/viewpost.jsp?thread=85551. 12

Vitousek, M. M., Siek, J. G., and Baker, J. (2014). Design and evaluation of gradual typing for python. 12

Wadler, P. and Findler, R. B. (2009). Well-typed programs can't be blamed. In Castagna [2009], pages 1–16. 25

Winther, J. (2011). Guarded type promotion: Eliminating redundant casts in Java. In *Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs*, FTfJP '11, pages 6:1–6:8. 118, 119, 121, 136, 146

Wolff, R., Garcia, R., Tanter, É., and Aldrich, J. (2010). Gradual featherweight typestate. Technical Report CMU-ISR-10-116, Carnegie Mellon University. 12

Wolff, R., Garcia, R., Tanter, É., and Aldrich, J. (2011). Gradual typestate. In Mezini [2011], pages 459–483. 106, 147, 177

Wright, A. K. and Cartwright, R. (1997). A practical soft type system for scheme. *ACM Trans. Program. Lang. Syst.*, 19(1):87–152. 11

Wrigstad, T., Nardelli, F. Z., Lebresne, S., Östlund, J., and Vitek, J. (2009). Integrating typed and untyped code in a scripting language. Technical report, Purdue University. 10, 13, 35

Zenger, M. and Odersky, M. (2005). Independently extensible solutions to the expression problem. In *In Proc. FOOL 12*. 125