On the Use of Type Predicates in Object-Oriented Software: The Case of Smalltalk *

Oscar Callaú[1] Romain Robbes[1] Éric Tanter[1] David Röthlisberger[2] Alexandre Bergel[1]

[1]PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile
{oalvarez,rrobbes,etanter,abergel}@dcc.uchile.cl

[2]School of Informatics and Telecommunications
Faculty of Engineering
Universidad Diego Portales
davidroe@mail.udp.cl

Abstract

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 these questions will assist practitioners and researchers with providing information about the state of the practice, and informing the active research program of retrofitting type systems, clarifying whether complex flow-sensitive typing approaches are necessary. Other areas, such as refactoring and teaching object orientation, can also benefit from empirical evidence on the matter.

We report on a study of the use of type predicates in a large base of over 4 million lines of Smalltalk code. Our study shows that type predicates are in fact widely used to do explicit type dispatch, suggesting that flow-sensitive typing approaches are necessary for a type system retrofitted for a dynamic object-oriented language.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Language; Design

Keywords Flow-sensitive typing; Object-oriented languages; Type predicates

* This work is partially funded by FONDECYT Projects 1110051, 1120094, 1140068 and 11110463.

DLS '14, October 20–24, 2014, Portland, OR, USA.

ACM 978-1-4503-3211-8/14/10. . . $15.00.
http://dx.doi.org/10.1145/2661088.2661091

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 [10, 18] 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 can inform both the general object-oriented programming community and the active research program of designing type systems for existing dynamic languages (*e.g.* [11, 13, 21, 31, 32, 34]), many of which are object oriented. Indeed, retrofitting a type system onto an existing language demands to properly accommodate the programming idioms embraced by programmers. Failing to do so compromises adoption of the retrofitted type system. Hence, informing about 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 [30, 31]. 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:

```
; 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.

[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 [16]—Scott Meyers.*

Knowing if the argument is a number is determined by the function number? (of type $Any \rightarrow Boolean$). In order to type this method, the type system must be able to understand that the application of add1 (of type $Number \rightarrow Number$) 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 [32], *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 [11, 21, 34].

Guha *et al.* [11] 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* [34], 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 [4]. Binary methods are indeed well-known to be hard to properly implement in an object-oriented language [7]. 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[2]. 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 paper, 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 [9]) would be "good enough".

RQ3: How prevalent is the use of logical combinations of type predicates? Logical types [32] 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 [11, 21, 34] or typestate checking [8, 27] are necessary.

Structure of the paper. In Section 2 we describe the experimental corpus and methodology as well as a classification of discovered predicates. The following four sections report on the four research questions above, respectively. Section 7 discusses the threats to validity, Section 8 reviews related work, and Section 9 concludes with recommendations for type system designers and practitioners.

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.

2.1 Corpus

We analyze a body of 1,000 projects, which we used previously in a study of the use of reflective features [5]. 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 (`http://www.squeaksource.com`) Smalltalk repository taken in early 2010. 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[3]. 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 [14], 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 [5] to statically

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

[3] 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.

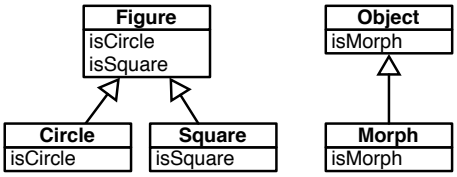

Figure 1. Examples of polymorphic type predicates.

trace the declarations and usages of type predicates in the software ecosystem[4].

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.

2.2.1 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"
```

2.2.2 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:

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

2.2.3 Polymorphic

Polymorphic type predicates are methods that play the role of type discriminators, just like string? in Racket. Figure 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).

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

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.

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.

2.2.4 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 [9]. 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.

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.

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 [4] 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.
- *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 [12, 19, 25, 33, 36] report that object-oriented software not only evolve by adding new classes, but

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 1. Usage categories of type predicates with their refinements.

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.

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: 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 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. To better understand why, we performed a dynamic analysis of the collections present in the standard Pharo Development Image (version 1.2.1 of Pharo, with the Seaside web framework, and related sub-projects). We analyzed all collections in the image to determine how many are strictly homogeneous (*i.e.* filled with objects of the exact same class). We found that out of 554,262 collections, 94.6% are strictly homogeneous. While the results are not representative of all the projects (only the default image and Seaside), and some predicates may discriminate on the state of the objects (as we explore in RQ4), the homogeneity of collections appears to be a good reason why type predicates are seldom used when operating over collections.

3.3 Refinement

For the remaining 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 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 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 decide to 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 we are missing. The results we report in this paper 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.

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 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* [24].

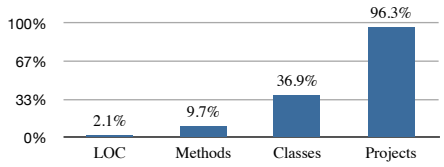

Figure 2. Presence of type predicates in LOC, methods, classes and projects.

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.

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.

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 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.

4.1 Predicate categories

Table 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 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.

4.2 Usages context and predicate categories

Table 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

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 2. Usages distributions for coarse and fine-grained 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 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%).

in all categories but structural (6.1%). Particularly, nil predicates are rarely used in a collection context (1.2%).

4.3 Nil predicate

Since nil-related predicates are so prevalent, we investigate them further. In Table 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 3), which makes it even more easy to apply a non-null type technique.

Tony Hoare's self-admitted "billion-dollar mistake"[5] is hence alive and well in Smalltalk code. On the upside, this is 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.

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 [31] with occurrence typing.

Additionally, from Table 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.

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.

[5] `http://tinyurl.com/hoare-mistake`

5. Prevalence of logical combinations

To assess the practical value of logical types [32] 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)
```

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 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.

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%).

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.

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.

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 [32] in that a full-blown flow-sensitive type system should account for logical combinations of predicates.

6. Prevalence of constant predicates

Object-oriented languages usually support mutable state. This means that predicate methods may not be constant. As a consequence, occurrence typing as originally formulated [31, 32] is unsound if predicates vary over time. Advanced approaches like flow typing [11, 21, 34] and typestates [8, 27] 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.

6.1 Classification of predicates

In order to assess the prevalence of the issue related to mutable state, we look at how polymorphic predicates are implemented.[6] 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`).[7] 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 remaining of this analysis, we use the terms *constant* and *variable* in the meaning described above. This classification is a safe under-estimation of the number of constant predicate implementations, as mentioned in Section 2.2.3. This means that we may qualify certain implementations as variable even though they are in fact constant. In Section 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,[8] 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.

6.2 Prevalence of constant predicates

Figure 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 underestimation, the results suggest that a good majority of the predicates are possibly variable.

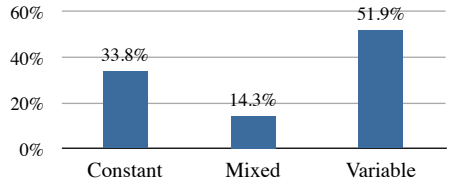

Figure 3. Predicates distribution based on constancy.

[6] 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 [5].

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

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

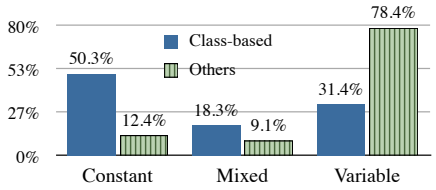

Figure 4. Refined constancy distribution, depending on predicate name.

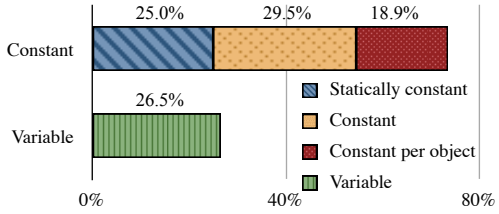

Figure 5. Refining constant and variable predicates with the dynamic analysis.

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 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.

6.4 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.

6.5 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 [26, 29]. 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 automatize. 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 [14]. 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 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[9], 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.

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.

[9] 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 7 for a wider discussion.

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 6.5. 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 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 6.5 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 [28]) 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.

8. Related Work

Tobin-Hochstadt and Felleisen [31] 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 [32] 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 [11]. 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 [6]. 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 [34], 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 [21–23] 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) [25]. 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 [15]. 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 than 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 [2]. 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 [3, 8, 27, 35], 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 [1, 35].

In a retrospective study of 10 open-source Java systems, Parnin *et al.* studied the adoption of Java generics by Java developers [20]; 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.

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 paper 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.

We hope these results motivate researchers to conduct related studies in other languages and scenarios in order to strengthen the confidence in these conclusions.

We foresee at least two directions for future work. First, we plan to extend the dynamic analysis to include more projects. For this, we could enhance the current dynamic analyzer to support more projects from the corpus. A more suitable alternative is to directly apply the dynamic analysis to runnable Smalltalk projects whose virtual images are available online. Second, we plan to apply the static analysis of this paper 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.

Acknowledgments. We thank the DLS reviewers for their helpful comments, and the European Smalltalk User Group (www.esug.org) for the sponsoring.

References

[1] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *Proceedings of Onward!*, pages 1015–1022. ACM, 2009.

[2] N. E. Beckman, D. Kim, and J. Aldrich. An empirical study of object protocols in the wild. In Mezini [17], pages 2–26.

[3] K. Bierhoff and J. Aldrich. 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, Oct. 2007. ACM Press. ACM SIGPLAN Notices, 42(10).

[4] J. Bloch. *Effective Java, 2nd Edition*. Addison-Wesley, 2008.

[5] O. Callaú, R. Robbes, É. Tanter, and D. Röthlisberger. How (and why) developers use the dynamic features of programming languages: the case of Smalltalk. *Empirical Software Engineering*, 2012. Online First.

[6] P. Chalin and P. R. James. 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, 2007.

[7] W. R. Cook. On understanding data abstraction, revisited. *ACM SIGPLAN Notices*, 44(10):557–572, 2009.

[8] R. DeLine and M. Fähndrich. Typestates for objects. In M. Odersky, 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, June 2004. Springer-Verlag.

[9] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In R. Crocker and G. L. Steele, Jr., editors, *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003)*, pages 302–312, Anaheim, CA, USA, Oct. 2003. ACM Press. ACM SIGPLAN Notices, 38(11).

[10] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[11] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In G. Barthe, editor, *Proceedings of the 20th European Symposium on Programming (ESOP 2011)*, volume 6602 of *Lecture Notes in Computer Science*, pages 256–275. Springer-Verlag, 2011.

[12] S. Krishnamurthi, M. Felleisen, and D. P. Friedman. 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, 1998. Springer-Verlag. ISBN 3-540-64737-6.

[13] G. Laforge. Whats new in Groovy 2.0? http://www.infoq.com/articles/new-groovy-20, 2012.

[14] M. Lungu, R. Robbes, and M. Lanza. 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, 2010.

[15] D. Malayeri and J. Aldrich. Is structural subtyping useful? an empirical study. In G. Castagna, editor, *Proceedings of the 18th European Symposium on Programming (ESOP 2009)*, volume 5502 of *Lecture Notes in Computer Science*, pages 95–111. Springer-Verlag, 2009.

[16] S. Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley, 2005.

[17] M. Mezini, editor. *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP 2011)*, volume 6813 of *Lecture Notes in Computer Science*, Lancaster, UK, July 2011. Springer-Verlag.

[18] O. Nierstrasz, S. Ducasse, and S. Demeyer. *Object-Oriented Reengineering Patterns*. Square Bracket Associates, 2009.

[19] B. C. Oliveira. Modular visitor components. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 269–293, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03012-3.

[20] C. Parnin, C. Bird, and E. R. Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *MSR 2011: Proceedings of the 8th International Working Conference on Mining Software Repositories*, pages 3–12, 2011.

[21] D. Pearce. Sound and complete flow typing with unions, intersections and negations. In R. Giacobazzi, J. Berdine, and I. Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7737 of *Lecture Notes in Computer Science*, pages 335–354. Springer Berlin Heidelberg, 2013.

[22] D. J. Pearce. 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, 2013. ISBN 978-1-4503-2042-9.

[23] D. J. Pearce and J. Noble. Implementing a language with flow-sensitive and structural typing on the JVM. *Electronic Notes in Theoretical Computer Science*, 279(1):47 – 59, 2011. ISSN 1571-0661. Proceedings of the Bytecode 2011 workshop, the Sixth Workshop on Bytecode Semantics, Verification, Analysis and Transformation.

[24] D. Posnett, V. Filkov, and P. Devanbu. Ecological inference in empirical software engineering. In *Proceedings of the 26th ACM/IEEE International Conference on Automated Software Engineering (ASE 2011)*, pages 362–371, 2011.

[25] R. Robbes, D. Röthlisberger, and É. Tanter. Extensions during software evolution: Do objects meet their promise? In J. Noble, editor, *Proceedings of the 26th European Conference on Object-oriented Programming (ECOOP 2012)*, Lecture Notes in Computer Science, pages 28–52, Beijing, China, June 2012. Springer-Verlag.

[26] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. *Theor. Pract. Object Syst.*, 3(4):253–263, Oct. 1997. ISSN 1074-3227.

[27] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[28] E. D. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. 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, 2010.

[29] A. Thies and E. Bodden. Refaflex: Safer refactorings for reflective java programs. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 1–11, 2012. ISBN 978-1-4503-1454-1.

[30] S. Tobin-Hochstadt. *Typed Scheme: From Scripts to Programs*. PhD thesis, Northeastern University, Jan. 2010.

[31] S. Tobin-Hochstadt and M. Felleisen. 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, Jan. 2008. ACM Press.

[32] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN Conference on Functional Programming (ICFP 2010)*, pages 117–128, Baltimore, Maryland, USA, Sept. 2010. ACM Press.

[33] M. Torgersen. 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, 2004.

[34] J. Winther. 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, 2011. ISBN 978-1-4503-0893-9.

[35] R. Wolff, R. Garcia, É. Tanter, and J. Aldrich. Gradual typestate. In Mezini [17], pages 459–483.

[36] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. In *In Proc. FOOL 12*, 2005.