Do Objects need Occurrence Typing?

Oscar Callaú

PLEIAD Laboratory, DCC, University of Chile
oalvarez@dcc.uchile.cl

***Abstract*—A major challenge of adding a type system to dynamic languages is to properly accommodate the programming idioms effectively used by programmers. For instance, *occurrence typing* was proposed to support flow-oriented reasoning in programs, that is, the common Scheme idiom of distinguishing the type of variables based on the results of type predicates. Considering object-oriented design principles, it would seem that manual dispatch based on explicit type checks should not occur often in practice. Therefore, when designing a type system for an existing object-oriented language, the question naturally raises of whether occurrence typing is at all useful, and if so, to what extent. This paper answers this question by studying the use of type predicates in a large base of Smalltalk code. Our study shows that type predicates are in fact widely used to do explicit type dispatch, suggesting that occurrence typing or similar flow-sensitive typing approaches are necessary.**

I. Introduction

Dynamic languages are typically favored for rapid prototyping, but as programs grow large and complex. Many efforts are targeted at providing retrofitted type systems for dynamic languages (*e.g.* [1], [2], [3], [4]). A major challenge in designing a retrofitted type system is to properly accommodate the most common programming idioms. Failing to do so means that the cost of adoption of the type system is likely to be deemed too high by programmers. The design of such a type system must therefore be directly informed by the practice of programming in the considered community.

Occurrence typing: A notable example of this approach is the Typed Racket effort (formerly Typed Scheme [5], [4]), a statically-typed variant of Racket, itself a dialect of Scheme. Typed Racket is based on the notion of *occurrence typing*. Occurrence typing allows the type system to account for the use of *type predicates*—simple functions—to distinguish the type of a variable. For instance, consider the following Scheme definition:

```
(define (f x)   ; x is a number or a string
  (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 $Any \rightarrow Boolean$). 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.

Type dispatch with objects: The object-oriented programming paradigm is supposed to free developers from manual dispatch based on explicit type predicates by relying on polymorphism. For instance, the following example:

```
ClassRoom>>inLecture: aPerson ``is a Teacher or a Student"
(aPerson isKindOf: Student)
    ifTrue: [aPerson listen] ifFalse: [aPerson talk]
```

The manual dispatch can be refactored to just call a polymorphic method declared in the root class, Person, and implemented in the leaves, Student and Teacher. This suggests that—at least in theory—the scenarios handled by occurrence typing are irrelevant in an object-oriented setting. Nevertheless, object-oriented languages usually provide operators to do runtime type checks, such as instanceof in Java or isKindOf: in Smalltalk. Their use is however strongly discouraged, precisely because it does not "fit the paradigm" [6], with the exception of binary equality methods [7]. But if occurrence typing is only helpful for equality methods, one could reasonably argue that it is unnecessary to integrate it in type systems for object-oriented languages.

Contributions. When designing a type system for a dynamic object-oriented language, the question of whether or not it is worthwhile to integrate occurrence types remains. In order to answer this question, we perform an empirical study of the use of type predicates in the dynamic object-oriented language Smalltalk. We statically analyze 1,000 open source Smalltalk projects(4 million LOC). Our study reveals if, and how, type predicates are used in practice, providing guidance in the design of type systems for object-oriented languages.

Concretely, we study the following research questions:

- RQ1: How prevalent is the use of type predicates to do explicit dispatch?
- RQ2: What are the different forms of type predicates used? Are some categories largely predominant?

II. Experimental Setup

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

A. Corpus

We analyze a body of 1,850 projects, which we used previously in a study of the use of reflective features [8]. 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 project corpus is a snapshot of the Squeaksource Smalltalk repository taken in early 2010. Squeaksource is the *de facto* source code repository for open-source development in the Squeak and Pharo communities. The corpus includes a total of 4,445,415 lines of code distributed between 47,720 classes and 652,990 methods.

In order to analyze the 1,000 projects, we extend our previous framework [8] to statically trace the declarations and usages of type predicates in the software ecosystem[1].

B. Finding Predicates and Their Usages

In Smalltalk we are interested in polymorphic methods (*e.g.* isString) and primitive type checks using isKindOf:, Smalltalk's equivalent of Java's instanceof, or some variants thereof. We distinguish three main categories of predicates, plus one of "special predicates", all described below.

a) Nominal and Structural: Nominal, this category corresponds to *nominal* type checks, *i.e.* related to the actual class of an object (isKindOf: and isMemberOf:) Additionally, we also count type checks performed through explicit class comparison, such as reference equality ==. *Structural*, Smalltalk supports *structural* type checks using respondsTo: or canUnderstand:.

b) Polymorphic: Polymorphic type predicates are methods that play the role of type discriminators, just like string ? in Scheme. This category of predicates is therefore user-extensible, and we need a heuristic to detect them. Following the Smalltalk naming conventions, we consider a type predicate any selector (*i.e.* method name) that follows the pattern isXxxx—the prefix is the verb is, followed by any camel-case suffix. We only consider selectors that do not take any argument. The body of a type predicate method should return a boolean. However, our analysis cannot ensure that a predicate returns a boolean, unless it is returned literally; still, we do exclude predicate candidates whose body is literally *not* a boolean—we found 79 of these, less than 1% of the 8,573 implementations of type predicates, which comforts our impression that our heuristic is valid.

c) Special Predicates: Smalltalk also provides a number of polymorphic predicates, that we treat separately. These are families of predicates checking the common cases of null references (isNil), empty collections (isEmpty), or the integer zero (isZero). Like for nominal predicates, there are alternative ways to check these conditions (*e.g.* using equality). Since these predicates are very common use cases (similar to common recursive base cases in a functional language), we grant them their own category.

III. RQ1: Prevalence of type predicates

We start by reporting on the results of our predicate detection algorithm, and then classify predicate usages in order to refine our analysis.

A. Basic statistics in Squeaksource

Our predicate detection algorithm identified 4,513 different predicates. This represents 1.8% of all selectors in the corpus. Because the boundary between type and state abstractions is fuzzy (typestate systems actually account for them [9], [10], [11], and typestate-oriented programming in fact fuses them [12], [13]), we include these predicates in the study.

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

Usage context	Usages	% Usages
Dispatch*	128,250	74.2%
Collections*	5,589	3.2%
Assertion*	22,932	13.3%
Forward	10,420	6.0%
Others	5,654	3.3%
Total	172,845	100%
Selected	156,771	90.7%

Table I
Usage categories of type predicates. (*, categories selected for the study.)

On the usage side, these predicates are used 172,845 times in our corpus, spread out in 984 out of the 1,000 projects we considered. Surprisingly, only 746 usages (0.43%) occur inside an equality method, suggesting that the recommendation of using type checks only in equality methods [7] is far from followed in practice. This already suggests that occurrence typing would not be helpful only for equality methods, in fact quite the contrary.

We actually do not include all 172,845 usages in our study, because some usages do not impact the flow of the program in a way directly observable to our simple static analysis. The next section introduces the classification of predicate usages on which our refinement is based.

B. 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. This corresponds to the classical examples where occurrence typing helps. *Collections*, the predicate is used to filter or test elements inside a collection, with select:, reject:, detect:, allSatisfy:, etc. An occurrence 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. From a typing point of view, this is similar to a conditional where the false branch raises an error. The next statement after the assertion can use the predicate assertion or negation fact. *Forward*, the predicate is used to define another predicate. *Others*, The catch-all category for usages that do not fit in any of the previous ones.

Table I shows the number of raw usages and their percentages. Unsurprisingly, simple conditional dispatch is the most common usage idiom, with three-quarter of overall usages. Second comes Assertions with 13.3%, showing that type predicates are often used in testing contexts, or in pre/post-conditions. The three other categories are relatively scarce.

For the remainder of this study, we keep the predicates classified as Dispatch, Collections, or Assertions, and exclude the predicates classified as Forward or Other. Taken together, the three usage categories we select comprise more than 90% 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 an occurrence type system.

C. Prevalence of predicate usages

After refinement, we are left with 156,771 usages of type predicates that directly affect the control flow of programs. We evaluate the presence of type predicate usages at different levels of granularity: projects (98.3%), classes(47.8%), methods(14.7%) and LOCs(3.5%).

In Smalltalk projects, we find that 98.3% of projects use type predicates, *i.e. not* using type predicates is the exception rather than the rule. We find at class-level that slightly less than half—47.8%—of the classes use type predicates as part of their implementation. This figure comforts the claim that programmers use type predicates quite commonly. At a finer-grained level, we find that 14.7% of the methods are using type predicates. Clearly, occurrence typing has the potential to provide more accurate type information in the control flow of more than one out of 7 methods. But perhaps the most telling figure is the finest-grained one, which is the density of type predicate usages per LOC. We find a density of 0.035 predicates per line of code, or 3.5%. In other words, one might expect to read around 30 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 support of these would have a practical impact on the daily work of programmers.

IV. RQ2: Prevalence of categories of type predicates

We are also interested in the prevalence of specific *categories* of type predicates, as described in Section II-B. If certain categories of type predicates are much more commonly used than others, this would allow one to make informed decisions about which are most important to support.

A. Predicate categories

Table II shows the distribution of each predicate category (nominal, structural, polymorphic) by usages among all projects by usages (Occ), LOC, and number of methods and classes. We clearly see the categories of predicates are not equally distributed. Polymorphic predicate take the lion's share at nearly 90% of the total (139,644 usages), nominal type predicates follow with 10% (15,650 usages), and structural type predicates only amount to less than 1% of the total usages (less than 1,500 usages overall). Structural type predicates are hence very seldom used.

Kinds	Occ.	% Occ.	% LOC	% Mth	% Cls
Nominal	15,650	10.0	0.35	1.5	8.7
Structural	1,457	0.9	0.03	0.2	1.6
Polymorphic	139,664	89.1	3.14	13.6	45.5
Nil	76,760	55.0	1.7	8.6	33.7
Empty	14,638	10.5	0.3	1.8	10.9
Zero	11,756	8.4	0.3	1.3	7.6
User Defined	36,510	26.1	0.8	3.7	17.9
All	156,771	100.0	3.5	14.7	47.8

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

B. Special predicates

Since polymorphic predicates are so prevalent, we investigate them further. In particular, we separate the special predicates—Nil, Empty, and Zero—from the user defined predicates. Table II (last four rows) shows that the Nil category consists of more than half of the polymorphic predicate usages (55%; 76,760 usages). If we look at the distribution of usages of nil predicates (see Table II), we note that 8.64% of all methods include a usage of a nil predicate (and a density of predicates per lines of code of 2%).

C. User-defined predicates

More than a quarter of polymorphic predicate usages (36,510) are usages of user-defined predicates. These type predicates are roughly half as prevalent as the Nil category alone, and account for 23.28% of all type predicate usages. If we combine these usages with the usages of nominal type predicates—nominal type predicates can be seen as polymorphic type predicates waiting to be implemented—, we arrive at 52,160 usages, or 33.27% of all usages; close to a third. This indicates a potential usefulness of a type system able to handle arbitrary type predicates, as in Typed Racket.

V. Related Work

In [14], 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. Our experiment further confirms that at least occurrence typing is 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 [3]. In 1,5 million LOC, they detect 13,500 occurrences of type testing operators. We detect proportionally much more occurrences, even without considering user-defined polymorphic predicates (about 3 times more). They use this measurement as a motivation for their work. Our study strengthens the argument that object-oriented programmers tend to use explicit type checks sufficiently enough to warrant specific support for them.

VI. 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. This work sheds light on the need to support explicit type-based reasoning in object-oriented programs, looking at a large Smalltalk codebase. 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 30 lines of code. Therefore, occurrence typing—in any of its possible forms—is definitely useful for objects.

- RQ2: Special predicates (Nil, Empty, Zero) account for almost two thirds of all usages. This suggests that a simpler, less general approach specifically tailored to these cases would already enjoy a broad applicability.

ACKNOWLEDGMENT

We thank ESUG (`esug.org`), the European Smalltalk User Group, for its financial support.

REFERENCES

[1] G. Bracha and D. Griswold, "Strongtalk: Typechecking Smalltalk in a production environment," in *Proceedings of the 8th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 95)*. Washington, D.C., USA: ACM Press, Oct. 1993, pp. 215–230, aCM SIGPLAN Notices, 28(10).

[2] C. T. Haynes, "Infer: A statically-typed dialect of Scheme," Indiana University, Tech. Rep. 367, 1995.

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

[4] 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)*. San Francisco, CA, USA: ACM Press, Jan. 2008, pp. 395–406.

[5] S. Tobin-Hochstadt, "Typed Scheme: From Scripts to Programs," Ph.D. dissertation, Northeastern University, Jan. 2010.

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

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

[8] 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.

[9] 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)*. Montreal, Canada: ACM Press, Oct. 2007, pp. 301–320, aCM SIGPLAN Notices, 42(10).

[10] R. DeLine and M. Fähndrich, "Typestates for objects," in *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, ser. Lecture Notes in Computer Science, M. Odersky, Ed., no. 3086. Oslo, Norway: Springer-Verlag, Jun. 2004, pp. 465–490.

[11] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Transactions on Software Engineering*, vol. 12, no. 1, pp. 157–171, 1986.

[12] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks, "Typestate-oriented programming," in *Proceedings of Onward!* ACM, 2009, pp. 1015–1022.

[13] R. Wolff, R. Garcia, É. Tanter, and J. Aldrich, "Gradual typestate," in *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP 2011)*, ser. Lecture Notes in Computer Science, M. Mezini, Ed., vol. 6813. Lancaster, UK: Springer-Verlag, Jul. 2011, pp. 459–483.

[14] S. Tobin-Hochstadt and M. Felleisen, "Logical types for untyped languages," in *Proceedings of the 15th ACM SIGPLAN Conference on Functional Programming (ICFP 2010)*. Baltimore, Maryland, USA: ACM Press, Sep. 2010, pp. 117–128.