Is Structural Subtyping Useful? An Empirical Study

Donna Malayeri **Jonathan Aldrich**

December 2009
CMU-CS-09-100

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Structural subtyping is popular in research languages, but all mainstream object-oriented languages use nominal subtyping. Since languages with structural subtyping are not in widespread use, the empirical questions of whether and how structural subtyping is useful have thus far remained unanswered. This study aims to provide answers to these questions. We identified several criteria that are indicators that nominally typed programs could benefit from structural subtyping, and performed automated and manual analyses of open-source Java programs based on these criteria. Our results suggest that these programs could indeed be improved with the addition of structural subtyping. We hope this study will provide guidance for language designers who are considering use of this subtyping discipline.

Keywords: Multiple inheritance, multiple dispatch, diamond inheritance

1 Introduction

Structural subtyping is popular in the research community and is used in languages such as O'Caml [15], PolyToil [6], Moby [11], Strongtalk [5], and a number of type systems and calculi (e.g., [7, 1]). In the research community, many believe that structural subtyping is beneficial and is superior to nominal subtyping. But, structural subtyping is not used in any mainstream object-oriented programming language—perhaps due to lack of evidence of its utility. Accordingly, we ask: what empirical evidence could show that structural subtyping can be beneficial?

Let us consider the characteristics that a nominally-typed program might exhibit that would indicate that it could benefit from structural subtyping. First, the program might systematically make use of a subset of methods of a type, with no nominal type corresponding to this method set. A particular such implicit type might be used repeatedly throughout the program. Structural subtyping would allow these types to be easily expressed, without requiring that the type hierarchy of the program change. This is particularly beneficial when a nominal hierarchy cannot be changed (due to lack of access to or control of the applicable source code), as changing a nominal hierarchy generally requires changes to the intended subtypes. For example, in Java, to express the fact that class C implements interface I, C's source must be modified.

Second, there might be methods in two different classes that share the same name and perform the same operation, but that are not contained in a common nominal supertype. There are a number of reasons why such a situation might occur, such as oversight on the part of the original designers. This is particularly likely when the original code did not need to make use of the implicit interface induced by these common methods. Alternatively, perhaps such a need did exist, but programmers resorted to code duplication rather than refactoring the type hierarchy—possibly because the source code was not accessible or could not be changed. On the other hand, with structural subtyping, the two classes in question would automatically share a common supertype consisting of the shared methods.

Or, programs might use the Java reflection method `Class.getMethod()` to call a method with a particular signature in a generic manner. For instance, we may wish to write a method m that can be passed as an argument any object that contains a "`String getName()`" method. In nominally typed languages, this can generally be achieved only through dynamic means such as reflection; in contrast, structural subtyping provides such a capability in a statically-checkable manner.

Finally, suppose a programmer is faced with the challenge of writing a class C that only supports a subset of its declared interface I. But, such a super-interface does not exist and cannot be defined, perhaps due to library use. One possible implementation strategy is simply throw an exception (e.g., `UnsupportedOperationException`) when one of C's unimplemented methods is called. In contrast, with structural subtyping, the intended structural super-interface could simply be used.

With all of these characteristics in mind, we performed several manual and automated analyses on (up to) 29 open-source Java programs. In the case of manual analyses, a subset of the subject programs was considered. Each of these analyses aimed to answer one question: are nominally-typed programs using implicit structural types? The result was that indeed they were; representing these types explicitly could therefore produce desirable characteristics, such as increased code reuse and decreased maintenance effort.

In the empirical evaluation, answers to the following questions were sought:

1. Does the body of a method use only a subset of the methods of its parameters? If so, structural types could ease the task of making the method more general. (Sect. 3)

2. If structural types are inferred for method parameters, do there exist inferred types that are used

repeatedly, suggesting that they represent a meaningful abstraction? (Sect. 3.3)

3. How many methods always throw "unsupported operation" exceptions? In such cases, the enclosing classes support a structural supertype of the declared class type; the latter contains all of the declared and inherited methods of the class (regardless of their implementation, or lack thereof). (Sect. 4)

4. What is the nature and frequency of *common methods*? That is, sets of methods with identical names and signatures, but that are not contained in any common supertype of the enclosing classes. (Sect 5.1)

5. How many common methods represent an accidental name clash? (Sect 5.2)

6. Can structural subtyping reduce some types of code duplication? (Sect. 5.3)

7. Is there empirical evidence of a potential synergy between structural subtyping and external methods? (Sect. 6)

8. Do programs use reflection where structural types would be preferable? (Sect. 7)

Thus, a variety of facets of existing programs were considered. While none of these aspects is conclusive on its own, taken together, the answers to the above questions provide evidence that even programs written with a nominal subtyping discipline could benefit from structural subtyping. This study provides initial answers to the above questions; further study is needed to fully examine all aspects of some questions, particularly questions 5 and 6. Additionally, one must bear in mind that structural subtyping is not always the appropriate solution; there do exist situations in which nominal subtyping is more appropriate.

To our knowledge, this is the first systematic corpus analysis to determine the benefits of structural subtyping. The contribution of this chapter are: (1) identification of a number of characteristics in a program that suggest the use of implicit structural types; and (2) results from automated and manual analyses that measure the identified characteristics.

2 Corpus and Methodology

For this study, the source code of up to 29 open-source Java applications were examined (version numbers of the applications are provided in Appendix A). The full set of subject programs were used for the automated analyses, while (while for practical considerations) manual analyses were performed on various subsets of these (ranging from 2 to 8 members). The applications were chosen from the following sources: popular applications on SourceForge, Apache Foundation applications, and the DaCapo benchmark suite.[1]

The full set of programs range from 12 kLOC to 161 kLOC, programs that were selected based on size, type (library/framework vs. sealed applications[2]) and domain (selecting for variety). For some of the manual analyses, we favored applications with which we were familiar (as this aided analysis), but we also aimed for variety in both application type and domain. All of the manual analyses, including the subjective analyses, were performed by one observer only—the author. The methodology for each analysis is described in the corresponding section; further details are available in Appendix A.

[1] http://dacapobench.org/

[2] Here we define a *sealed application* as a complete program that is not intended to be directly reused.

3 Inferring Structural Types for Method Parameters

It is considered good programming practice to make parameters as general as the program allows. Bloch, for example, recommends favoring interfaces over classes in general—particularly so in the case of parameter types [3]. An analogous situation arises in the *generic programming* community, where it is recommended that generic algorithms and types place as few requirements as possible on their type parameters (e.g., what methods they should support) [21].

Bloch acknowledges that sometimes an appropriate interface does not exist. For example, class java.util.Random implements only one (empty) marker interface. In such a case the programmer is forced to use classes for parameter types—even though it is possible that multiple implementations of the same functionality could exist [3]. This is a situation where structural subtyping could be beneficial, as it allows programmers to create supertypes after-the-fact.

As it is impossible to retroactively implement interfaces in Java, we hypothesized that method parameter types are often overly specific, and sought to determine both (1) the degree and (2) the character of over-specificity. To answer question (1), an automated whole-program analysis to infer structural types for method parameters was performed. Methodology and quantitative results are described in Sect. 3.1. To properly interpret this data, however, we must consider question (2). Accordingly, the inferred structural types from the previous analysis were manually examined and the following qualitative question was considered: would changing a method to have the most general structural type potentially improve the method's interface (Sect. 3.2)? Across all applications, the occurrences of inferred structural types that were supertypes of classes and interfaces of the Java Collections Library were enumerated. Of these, in Sect. 3.3 those structural types that a client might plausibly wish to implement while *not* simultaneously implementing a more specific nominal type (e.g., Collection, Map, etc.) are presented.

3.1 Quantitative Results

The analysis infers structural types for method parameters, based on the methods that were actually called on the parameters. (For example, a method may take a List as an argument, but may only use the add and iterator methods.) The analysis, a simple inter-procedural dataflow analysis, re-computes structural types for each parameter of a method until a fixpoint is reached. Structural types were not inferred for calls to library methods (for modularity purposes), nor were they inferred for primitive types, common types such as String and Object, and cases where the inferred structural type would have a non-public member. Finally, to simplify the analysis, structural types were *not* inferred for objects on the left-hand side of an assignment expression.

The analysis is conservative; in the case where a parameter is not used (or only methods of class Object are used), no structural type is inferred for it. A parameter may be unused because (a) it is expected that overriding methods will use the parameter, or (b) because the method may make use of the parameter when the program evolves, or (c) because it is no longer needed, due to changes in the program. In the case of method overriding, the analysis ensures that the same structural types are inferred for corresponding parameters in the entire method family.

The first set of results appear in Table 1. In Ant, 9.7% of parameters were unused, 47.2% of parameters had a primitive type, were String, or were Object. For 0.3% of parameters, a call was made to a non-public method, which means that a structural type could not be used in this case (as the visibility of all members of a structural interface must be public). 1.0% of parameters could not have a structural type inferred due to the fact that the associated method was overriding a method in a library. Finally, for 15.8% of parameters, a structural type could not be inferred, due to the fact that the parameter was assigned to a

	LOC	Unused	Primitive type	Non-public call	Library override	Called library	Assigned	% Inferrable of total	% Inferrable of candidates
Ant	62k	9.7%	47.2%	0.3%	1.0%	9.1%	15.8%	17.0%	40.5%
antlr	42k	16.7%	50.2%	0.4%	0.2%	8.1%	9.0%	15.5%	47.6%
Apache coll	26k	8.4%	55.0%	2.1%	6.0%	1.2%	16.4%	11.0%	38.4%
Areca	35k	9.7%	39.4%	0.1%	4.8%	9.3%	20.5%	16.1%	35.1%
Cayenne	95k	8.3%	47.0%	0.5%	3.1%	8.1%	11.2%	21.9%	53.2%
Columba	70k	11.6%	40.8%	0.6%	19.8%	5.5%	9.0%	12.6%	46.4%
Crystal	12k	18.0%	4.1%	0.2%	17.9%	7.4%	22.3%	30.1%	50.3%
DrJava	59k	13.5%	42.5%	0.8%	13.2%	7.8%	7.8%	14.3%	47.9%
Emma	23k	20.5%	42.3%	0.4%	0.9%	7.4%	8.8%	19.6%	54.6%
freecol	62k	8.7%	38.5%	0.0%	11.5%	3.9%	11.8%	25.5%	61.9%
hsqldb	62k	14.4%	61.3%	6.4%	3.9%	1.0%	4.8%	8.2%	58.5%
HttpClient	18k	14.3%	55.7%	0.1%	0.3%	5.7%	5.1%	18.8%	63.5%
jEdit	71k	11.8%	56.9%	1.0%	9.7%	4.9%	8.5%	7.2%	35.1%
JFreeChart	93k	8.1%	45.9%	0.4%	1.4%	14.3%	10.4%	19.6%	44.2%
JHotDraw	52k	18.3%	32.3%	0.0%	7.7%	11.1%	10.1%	20.5%	49.2%
jruby	86k	19.7%	27.2%	0.4%	0.8%	3.0%	16.0%	32.9%	63.4%
jung	26k	8.1%	33.8%	0.1%	4.8%	22.9%	12.0%	18.2%	34.3%
LimeWire	97k	13.7%	45.8%	1.4%	7.1%	7.9%	6.7%	17.5%	54.5%
log4j	13k	12.3%	46.8%	0.7%	4.7%	6.6%	10.0%	18.8%	53.1%
Lucene	24k	12.3%	58.3%	0.6%	0.1%	4.9%	14.8%	9.2%	31.8%
OpenFire	90k	14.0%	39.7%	0.2%	6.1%	7.6%	10.9%	21.4%	53.5%
plt collections	19k	15.8%	19.5%	0.3%	3.0%	6.8%	42.1%	12.4%	20.3%
pmd	38k	31.3%	32.7%	0.0%	1.3%	6.5%	8.4%	19.7%	56.9%
poi	50k	15.9%	69.8%	0.7%	3.3%	1.3%	2.1%	6.7%	66.2%
quartz	22k	15.4%	54.2%	0.0%	0.8%	5.9%	5.6%	18.2%	61.2%
Smack	40k	17.2%	45.3%	0.2%	1.6%	12.5%	8.1%	15.1%	42.2%
Struts	28k	6.3%	58.1%	0.1%	4.4%	5.1%	18.9%	7.1%	22.8%
Tomcat	126k	13.6%	54.6%	0.1%	3.2%	3.7%	11.0%	13.8%	48.3%
xalan	161k	10.5%	56.5%	1.3%	2.7%	2.5%	10.9%	15.7%	54.1%
Average		**13.7%**	**44.9%**	**0.7%**	**5.0%**	**7.0%**	**12.0%**	**16.7%**	**47.9%**

Table 1: Categories of method parameters when running structural type inference over 29 programs. “Unused” denotes the percentage of parameters that were not transitively used in the program, “primitive type” is the percentage of parameters that were either a primitive type, or were String or Object, “non-public call” is the percentage of parameters on which a non-public method was called (in which case a structural type could not be inferred), and “library override” is the percentage of paramters for which a structural type could not be inferred due to the fact that the method was an override of a library method. “Called library” is the percentage of parameters for which a structural type could not be inferred because a library method was transitively called and “assigned” is the percentage of parameters that were assigned to a local or member variable and did not have structural types computed. “Percent inferrable of total” is the percentage of all parameters that could have a structural type inferred, while “percent inferrable of candidates” is the percentage of inferrable parameters, when considering only those parameters for which a structural type would be meaninful.

local variable or member variable (this was a limitation of the analysis).

Considering all parameters, an average of 16.7% could have a structural type inferred. However, if we exclude parameters that fall into the categories in columns 3–6 (i.e., unused parameters, primitive types, non-public calls, and library overrides), then an average of 47.9% of parameters could have a structural type inferred. This second figure is more relevant, as it is not meaningful to infer structural types for parameters that fall into the aforementioned categories.

The analysis also computed some characteristics of these structural types that were inferred; results are displayed in Table 2. An average of 94.0% of parameters were declared with an overly precise nominal type (i.e., the nominal type contained more methods than were actually needed). For an average of 91.8% of the inferred parameters a corresponding nominal type did *not* exist in the program that would make the parameter type as *general* as possible (i.e., a nominal type that contained only those methods transitively called on the object). There were an average of 3.7 methods in the inferred structural types, across all programs, while there were an average of 41.7 methods in the corresponding nominal types. Finally, there was an average median of 1.2 structural types inferred for each nominal type in the program, and an average maximum of 23.4 structural types.

Note that the data shows that inferred structural types do not have many methods,[3], while the corresponding nominal types have quite a few methods. This shows that there is quite a large *degree* of over specificity—more than a full order of magnitude—in addition to the large percentage of overly specific parameters. This is likely due to the overhead of naming and defining nominal types, as well as the lack of retroactive interface implementation. The analysis also showed that when nominal types were as general as possible, they had very few members—one or two on average. This is in accordance with previous work which found that interfaces are generally smaller than classes [23].

Additional data. For a given nominal type, there were not many corresponding structural types (2.5 on average, a median of 1.2). The data followed a power law distribution, with an average maximum of 24; that is, small values were heavily represented, but there were also a few large values. The low median suggests that the overhead of naming structural types is not necessarily high; it is plausible that programmers would be able to name and use structural types for around half of the nominal parameter types.

Finally, if we were to define new interfaces everywhere possible, the average increase in the number of interfaces is 313%, the median is 287%, and the maximum is 1000%. This illustrates the infeasibility of defining new nominal types for the inferred structural types. Note that only those interfaces for which the implements clause of a class could be modified (i.e., those classes in the program's source) were considered; in general, the situation is even worse, as programmers may wish to define new supertypes for types contained in libraries.

3.2 Qualitative Results

Though the results show that many parameters are overly specific, it is not necessarily a good design to make every parameter as general as possible. This is because a method might be currently only using a particular set of methods, but later code modifications may make it necessary to use a larger set; a more general type could hinder program evolution. On the other hand, more general types make methods

[3]There is one outlier in the data; in pmd, inferred structural types had 29.5 methods on average. This is due to the use of the visitor design pattern—all visit methods are accessible from the top visitor accept method, since each override calls a specific visit method.

	LOC	% Inferrable	% Overly specific	% Structural needed	Avg methods/ structural type	Avg methods/ nominal type	Struct types/nominal	
							median	max
Ant	62k	40.5%	98.6%	97.7%	2.1	36.0	1	27
antlr	42k	47.6%	100.0%	98.8%	2.2	14.0	2	9
Apache coll	26k	38.4%	89.5%	83.0%	2.0	16.0	1	11
Areca	35k	35.1%	99.1%	97.4%	2.8	35.9	1	35
Cayenne	95k	53.2%	96.3%	92.6%	2.4	31.3	2	27
Columba	70k	46.4%	99.6%	98.8%	1.9	51.6	1	19
Crystal	12k	50.3%	98.8%	96.6%	3.2	15.7	1	19
DrJava	59k	47.9%	89.5%	87.1%	3.2	56.3	1	20
Emma	23k	54.6%	88.2%	87.8%	3.4	17.1	1	9
freecol	62k	61.9%	98.6%	97.9%	2.6	84.8	1	57
hsqldb	62k	58.5%	99.4%	99.4%	1.7	48.9	2	34
HttpClient	18k	63.5%	96.3%	94.8%	3.5	27.0	1	17
jEdit	71k	35.1%	95.5%	95.5%	2.2	119.6	1	20
JFreeChart	93k	44.2%	97.7%	93.5%	3.3	53.9	1	35
JHotDraw	52k	49.2%	100.0%	97.2%	3.0	57.0	2	19
jruby	86k	63.4%	98.1%	97.5%	6.9	66.1	1	85
jung	26k	34.3%	96.3%	88.3%	1.8	32.1	1	15
LimeWire	97k	54.5%	98.5%	94.9%	2.1	34.6	1	21
log4j	13k	53.1%	96.5%	95.0%	2.3	56.7	1	6
Lucene	24k	31.8%	80.5%	77.4%	1.6	13.5	1.5	8
OpenFire	90k	53.5%	99.4%	96.7%	2.4	37.1	1	45
plt collections	19k	20.3%	58.2%	59.3%	1.5	39.8	1	25
pmd	38k	56.9%	72.9%	69.1%	29.5	48.2	2	23
poi	50k	66.2%	88.0%	87.0%	1.9	22.8	1	8
quartz	22k	61.2%	100.0%	99.1%	2.2	36.6	1	11
Smack	40k	42.2%	100.0%	91.6%	4.4	29.2	1	13
Struts	28k	22.8%	96.4%	96.4%	2.1	32.4	1	13
Tomcat	126k	48.3%	96.8%	96.3%	4.5	37.6	2	32
xalan	161k	54.1%	96.4%	96.0%	5.3	56.5	1	16
Average		**47.9%**	**94.0%**	**91.8%**	**3.7**	**41.7**	**1.2**	**23.4**

Table 2: Results of running structural type inference. *Percent inferrable* is the percentage of candidate parameters that could have a structural type inferred (i.e., last column in Fig. 1), *percent overly specific* is the percentage of the inferrable parameters that have an overly specific nominal type, *percent structural needed* is the percentage of the inferrable parameters for which a most general nominal type does *not* exist, *average methods per structural type* is the average number of methods in the inferred structural types, *average methods per nominal type* is the average number of methods in nominal types that appear as parameter types (including inherited methods), and *median/maximum structural types per nominal* are the median and maximum, respectively, of the number of inferred structural types corresponding to each nominal type.

more reusable, which aids program evolution. For this reason, a refactoring to structural types (or even structural type inference) cannot be a fully automated process—programmers must consider each type carefully, keeping in view the kinds of program modifications that are likely to occur. Additionally, for some structural types, there may ever be only one corresponding nominal type, in which case using a structural type is of limited utility.

Accordingly, an empirical question was considered: would changing a given method to have the most general structural types for its parameters make the method more general in a way that could improve the program? To determine this, we inspected each method and asked two questions. First, does the inferred parameter type S generalize the abstract operation performed by the method, as determined by

the method name? Second, does it seem likely that there would be multiple subtypes of *S*?

Two applications were studied: Apache Collections (a collections library) and Crystal (a static analysis framework). Of methods for which a structural type was inferred on one or more parameters, we found that 58% and 66%, respectively, would be generalized in a potentially useful manner if the inferred types were used.

For example, in Apache Collections, in the class OnePredicate (a predicate class that returns true only if one of its enclosing predicates returns true), the factory method getInstance(Collection) had the structural type {iterator(); size();} inferred for its parameter. This would make the method applicable to any collection that supported only iteration and retrieving the collection size, even if it didn't support collection addition and removal methods. There were 25 other methods in the library that used this structural type. Another example is the method ListUtils.intersection which takes two List objects. However, the first List need only have a contains method, and the second List need only have an iterator method (for this latter parameter, the interface Iterable could be used). There were also 8 methods that took an Iterator as a parameter, but never called the remove method. With a structural type for the method, the type would clearly specify that a read-only iterator can be passed as an argument.

In Crystal, two methods took a Map parameter that used only the get and put methods. Converting the method to use this structural type would make it applicable to a map that did not support iteration (such a type exists in Apache Collections, for example). Also, there were 11 methods that use only the methods getModifiers() and getName() on an IBinding object (an interface in the Eclipse JDT). Replacing the nominal type with a structural type would allow the program to substitute a different "bindings" class that supported only those two methods.

Of course, for some of these structural types, there may not be a large number of classes that implement its methods but not all of the methods of a more specific nominal type, e.g., Collection. However, we believe that all of the aforementioned types represent meaningful abstractions. Furthermore, since it is conceivable that a programmer may define a class implementing that abstraction, using these more general types would increase the applications' reusability.

3.2.1 Translation to Whiteoak

Using the inference algorithm, we also developed an automated translation of programs from Java to Whiteoak [14], a research language that extends Java with support for structural subtyping. we performed this translation on two programs: Apache Collections and Lucene, confirming the correctness of the analysis and demonstrating its practical use.

3.3 Uses of Java Collections Library

We next considered inferred structural types that were supertypes of interfaces and classes in the Java Collections Library. Over all applications, there were 67 distinct types in total, though not all appeared to express an important abstraction. We made a conservative subjective finding that at least 10 of these types *were* potentially useful; these are displayed in Table 3, along with a description of possible implementations. For instance, there were 168 inferred parameters that used only the get() and containsKey() methods of Map. It would be useful to have a type corresponding to this abstraction, particularly if the map is immutable and must have its contents set at creation-time. A type consisting of these two methods would also be useful to support the pattern that once a map is populated, clients should not make modifications.

The relatively high number of occurrences of each of these structural types suggests their utility, even

Methods in type	Uses	Description
`get(Object); containsKey(Object);`	168	Read-only non-iterable map; for instance, a read-only hashtable[4]
`iterator(); isEmpty(); size();`	114	Read-only iterable collection that knows its size; for instance, a read-only list
`add(Object); addAll(Collection);`	101	Write-only collection; for instance, a log
`put(Object, Object);`	55	Write-only map
`hasNext(); next();`	28	Read-only iterator
`contains(Object);`	21	Read-only collection that does not support iteration; for instance, a read-only hashset
`get(Object); put(Object, Object);`	15	Non-iterable map; for instance, a hashtable
`contains(Object); iterator(); size();`	11	Read-only iterable collection that knows its size and can be polled for the existence of an element; for instance, an iterable hashset
`add(Object); contains(Object); iterator(); size();`	10	Same as above, but that also supports adding elements
`iterator(); size(); toArray(Object[]);`	8	Read-only collection that can be converted to an array; for instance, a read-only array

Table 3: Uses of Java Collections classes across 29 programs, as inferred using the parameter structural type inference. (Erasures are used in lieu of generic types.)

though the types contain few methods. It further shows that programs routinely make use of types that the library designers either did not anticipate or chose not to support.

In summary, the data shows that programs make repeated use of many implicit structural types. A language that would allow defining these types explicitly could be beneficial, as it can help programmers make their methods more generally applicable.

3.4 Related work

Forster [12] and Steimann [22] have described experience using the Infer Type refactoring, which generates new interfaces for inferred types and replaces uses of overly specific types with these interfaces. This analysis is more general than the one used here, because it considers all type references, not just parameter types. However, the refactoring is limited by the fact that classes in libraries cannot retroactively implement new interfaces. Steimann found that when applying this refactoring, the number of total interfaces almost quadrupled—an increase of 369%.[5] In his analysis, there were an average of 2.8 and 4.5 reference per inferred type, respectively, in the two studied applications, DrawSWF and JHotDraw. By comparison, in DrawSWF, we found that structural types were used in an average of 2.7 parameters; for JHotDraw this value was 3.3, which differs from Steimann's result. This discrepancy is likely due to the fact that he considered types other than those of parameters. Additionally, both Forster and Steimann found that the number of variables typed with each new inferred interface followed a power law distribution, which is what we also found for parameters.

3.4.1 Summary of results

In summary, the parameter analysis suggests that there are many nominal types that could be made more general using structural subtyping, and most of these were qualitatively determined to be useful. Also,

[5] This differs slightly from our average of 313%, though this difference is likely due to the fact that Steimann considered only two applications.

	Number of classes
Read-only `Iterator`	50
Read-only `Collection`	19
Read-only `Map`	9
Read-only `Map.Entry`	6
Read-only `ListIterator`	6
`Collection` supporting everything but removal	5
`Map` supporting everything but removal	4
`Collection` supporting only read and removal methods	1
`Collection` supporting iteration, addition, and size only	1
`ListIterator` supporting read, add, and remove (but not `set()`)	1
`ListIterator` supporting only read and `set()` operation	1
`Map` supporting read, put, and size only	1
`Map` supporting read and put, but not size or removal	1
`Map` supporting everything but `entrySet()`, `values()` and `containsValue()`	1

Table 4: A selection of the structural interfaces "implemented" by classes in the subject programs once methods unconditionally throwing an UnsupportedOperationException are removed. (Actual method sets are omitted to conserve space.)

the inferred structural types had an order of magnitude fewer methods than the corresponding nominal types. It is infeasible to define new nominal types to correct this, due to the number of structural types inferred per nominal type and the resulting percentage increase in interfaces.

4 Throwing "Unsupported Operation" Exceptions

In the Java Collections Library, there are a number of "optional" methods whose documentation permits them to always throw an exception. This decision was due to the practical consideration of avoiding an "explosion" of interfaces; the library designers mentioned that at least 25 new interfaces would be otherwise required [18].

To determine if such super-interfaces would be useful in practice, the methods in the subject programs that unconditionally throw an UnsupportedOperationException were totalled. The program that had the most such methods was Apache Collections: there were 148 methods that unconditionally throw the exception (out of 3669 total methods, corresponding to 4%). Next, those methods that were overriding a method in the Java Collections Library were considered. To encode these optional methods directly would require 18 additional interfaces. There are only 27 interfaces defined in the library, so this represents a 67% increase. Note that this is a conservative estimate, as interactions between classes (e.g., an Iterable returning a read-only Iterator) were not considered. A selection of these structural super-interfaces is summarized in Table 4. For instance, there were 50 iterator classes that did not support the remove() operation, and 19 subclasses of Collection that supported a read-only interface.

Note that, with the exception of the read-only iterator, the sets of interfaces in Tables 4 and 3 are distinct from one another (though some are subtypes). This is likely due to the fact that different applications use different subsets of the methods of a class.

Structural subtyping could be helpful for statically ensuring that "unsupported operation" exceptions cannot occur, as it would allow programmers to express these super-interfaces directly.

5 Common Methods

In my experience, there are situations where two types share an implicit common supertype, but this relationship is not encoded in the type hierarchy. For example, suppose two classes both have a getName method with the same signature, but there does not exist a supertype of both classes containing this method. We call getName, and methods like it, *common methods*. Common methods can occur when programmers do not anticipate the utility of a shared supertype or when two methods have the same name, but perform different operations; e.g., Cowboy.draw() and Circle.draw() [16].

Accordingly, this section aims to answer three questions: (1) how often do common methods occur, (2) how many common methods represent an accidental name clash, and (3) do common methods result in code clones?

5.1 Frequency

A simple whole-program analysis to count the number of common methods in each application was performed. Only public instance methods were considered (resulting in slightly different data than that previously presented [17]). Results are in Table 5. Overall, common methods comprise an average of 19% of all public instance methods. That is, for 19% of methods, there existed another method with the same name and signature and the method was not contained in a common supertype of the enclosing types.

The number of types that share at least two common methods with another type was also computed; there were an average of 9% of such types. These are the cases in which a structural supertype is most likely to be useful. This high percentage indicates that there are a number of implicit structural types in most applications.

For example, in Apache Collections, UnmodifiableSortedMap and OrderedMap share the methods firstKey() and lastKey(). And, AbstractLinkedList and SequencedHashMap share the methods getFirst() and getLast(). Finally, BoundedMap and BoundedCollection have the common methods isFull() and maxSize().

In Lucene, a document indexing and search library, RAMOutputStream and RAMInputStream both support the seek(), close(), and getFilePointer() methods, which might be useful to move to a supertype. Also, the classes PhraseQuery and MultiPhraseQuery both support the methods add(Term), getPositions(), getSlop(), and setSlop(int).

5.2 Accidental Name Clashes

Of course, to interpret this data, we must consider cases where the common methods have the same *meaning*, and where callers are likely to call the methods with the same purpose in mind. If two methods have the same meaning, it might be useful to define a structural type consisting of that method. Two methods are defined as "having the same meaning" if they perform the same abstract operation, taking into account (a) the semantics of the method, and (b) the semantics of the enclosing types. This determination was made by examining the source code, using javadoc where available.

Two applications were studied: Apache Collections and Lucene. In Collections, under condition (a), there were no methods that had the same signature but performed different abstract operations. However, there were 2 cases (1% of all common methods) where the methods had the same meaning, but the enclosing classes did not appear to be semantic subtypes of some common supertype containing that method; i.e., condition (b) was not satisfied. For example, the classes ChainedClosure and

	LOC	Number of types	Types with >1 common method	Percentage	% common methods	Avg # classes/ common signature
Ant	62k	945	65	6.9%	31.3%	3.7
antlr	42k	226	26	11.5%	23.6%	2.7
Apache Collections	26k	550	19	3.5%	7.3%	2.7
Areca	35k	362	30	8.3%	15.4%	2.7
Cayenne	95k	1415	104	7.3%	18.1%	2.8
Columba	70k	1232	48	3.9%	17.3%	3.1
Crystal	12k	211	4	1.9%	5.1%	2.9
DrJava	59k	927	65	7.0%	12.1%	2.6
Emma	23k	443	22	5.0%	18.7%	3.4
freecol	62k	569	55	9.7%	20.6%	2.7
hsqldb	62k	355	31	8.7%	19.5%	2.6
HttpClient	18k	231	19	8.2%	15.0%	2.6
jEdit	71k	880	40	4.5%	11.7%	2.5
JFreeChart	93k	789	301	38.1%	39.5%	3.9
JHotDraw	52k	616	59	9.6%	19.0%	2.8
jruby	86k	997	83	8.3%	15.6%	3.1
jung	26k	531	24	4.5%	19.3%	2.4
LimeWire	97k	1689	88	5.2%	17.7%	3.1
log4j	13k	201	4	2.0%	13.6%	2.4
Lucene	24k	398	21	5.3%	13.4%	2.6
OpenFire	90k	1039	110	10.6%	19.0%	3.0
plt collections	19k	812	60	7.4%	7.5%	2.8
pmd	38k	478	24	5.0%	12.0%	2.7
poi	50k	539	62	11.5%	20.9%	2.6
quartz	22k	158	24	15.2%	20.0%	2.4
Smack	40k	847	115	13.6%	23.5%	3.3
Struts	28k	609	158	25.9%	45.2%	2.7
Tomcat	126k	1727	234	13.5%	32.6%	3.6
xalan	161k	1223	94	7.7%	16.1%	2.9
Average				**9.3%**	**19.0%**	**2.9**

Table 5: Common methods for each application. *Number of types* indicates the total number of types in the application, *types with greater than one common method* is the number of types that share more than one common method, *percentage* is the percentage of this compared to the total number of types, *percent common methods* is the percentage of public instance methods that is a common method, and *average number of classes per common signature* is the average number of classes for each common method signature.

SwitchClosure both had a getClosures() method, but ChainedClosure calls each of these closures in turn, while SwitchClosure calls that closure whose predicate returns true.

In Lucene, there were 42 instances of methods that had the same signature, but did not have the same meaning (19% of all common methods). In 32 of these cases, the methods were actually performing a different abstract operation. For example, HitIterator.length() returned the number of hits for a particular query, while Payload.length() returned the length of the payload data. An additional 10 cases did not satisfy condition (b) above. For example, in a high-level class IndexModifier, there were several cases where a method *m* performed some operation, then called IndexWriter.*m*, the latter performing a lower-level operation. So, the semantics of the methods were similar, but the semantics of each class was different.

Overall, the data is promising, as it indicates that most common methods have the same meaning and would benefit from being contained in a structural supertype—90% on average, across both appli-

```
private InlineMethodRefactoring(ICompilationUnit unit,
 MethodInvocation node, int offset, int length)
{
    this(unit, (ASTNode)node, offset, length);
    fTargetProvider= TargetProvider.create(unit, node);
    fInitialMode= fCurrentMode= Mode.INLINE_SINGLE;
    fDeleteSource= false;
}

private InlineMethodRefactoring(ICompilationUnit unit,
 SuperMethodInvocation node, int offset, int length)
{
    ...   // same method body as above
}
```

Figure 1: Example of code duplication in the Eclipse JDT. Structural subtyping could eliminate this duplication.

cations. Structural subtyping would allow these methods to be called in a generic manner, without the need to create additional interfaces.

5.3 Code Clones

We hypothesized that common methods can lead to code clones, as there is a common structure that is not expressed in the type system. To determine this, two applications were examined: Eclipse JDT and Azureus.

In the Eclipse Java Development Tools (JDT), the classes FieldAccess and SuperFieldAccess have no superclass other than Expression. The same problem occurs with MethodInvocation and SuperMethodInvocation, and ConstructorInvocation and SuperConstructorInvocation. We found 44 code clones involving these types (though some were only a few lines long). An example of a code clone involving MethodInvocation and SuperMethodInvocation appears in Fig. 1. Another code clone involving SuperConstructorInvocation and ConstructorInvocation appears in Fig. 2.

Similarly, in the Eclipse SWT (Simple Windowing Toolkit), there are 13 classes (such as Button, Label, and Link) with the methods getText and setText that get and set the main text for the control. But, there is no common IText interface. Azureus, a BitTorrent client, is an application that requires the ability to call these methods in a generic fashion. Azureus is localized for a number of languages, which can be changed at runtime. Accordingly, there are several instances of code similar to that of Fig. 3.

Note that some of this code duplication might be avoided if the class hierarchy were refactored. Obviously, this is not always possible—e.g., Azureus cannot modify SWT.

The code duplication in these examples can be dramatically reduced by taking advantage of structural types. For example, Fig. 4 shows how the code block of Fig. 3a could be re-written with structural types in the Unity language [17].

In summary, common methods can lead to undesirable code duplication. Structural subtyping can help eliminate this problem, without refactoring the class hierarchy.

```
case ASTNode.SUPER_CONSTRUCTOR_INVOCATION: {
   SuperConstructorInvocation superInvocation= (SuperConstructorInvocation) parent;
   IMethodBinding superBinding= superInvocation.resolveConstructorBinding();
   if (superBinding != null) {
      return getParameterTypeBinding(node, superInvocation.arguments(), superBinding);
   }
   break;
}
case ASTNode.CONSTRUCTOR_INVOCATION: {
   ConstructorInvocation constrInvocation= (ConstructorInvocation) parent;
   IMethodBinding constrBinding= constrInvocation.resolveConstructorBinding();
   if (constrBinding != null) {
      return getParameterTypeBinding(node, constrInvocation.arguments(), constrBinding);
   }
   break;
}
```

Figure 2: Code duplication involving SuperConstructorInvocation and ConstructorInvocation. Only the highlighted lines of code differ in the two blocks.

```
if (widget instanceof Label)
   ((Label) widget). setText(message);
else if (widget instanceof CLabel)
   ((CLabel) widget). setText(message);
else if (widget instanceof Group)
   ((Group) widget). setText(message);
... // 5 more items
```

(a)

```
if (widget instanceof CoolBar) {
   CoolItem[] items = ((CoolBar)widget).getItems();
   for(int i = 0; i < items.length; i++) {
      Control control = items[i].getControl();
      updateLanguageForControl(control);
   }
} else if (widget instanceof TabFolder) {
   ... // same code as highlighted above
} else if (widget instanceof CTabFolder) {
   ... // same code as highlighted above
... // 5 more items
```

(b)

Figure 3: Code excerpts from Azureus, illustrating an awkward coding style and duplication.

6 Cascading instanceof Tests

We considered the question of whether structural subtyping could provide benefits if used in conjunction with other language features—external methods in particular. *External methods* (also known as open classes) are similar to ordinary methods and provide the the usual dispatch semantics, but can be implemented outside of a class's definition, providing more flexibility. Multimethods are a generalized form of external method, defined outside all classes and allowing dispatch on any subset of a method's arguments [9, 4, 10, 17].

Since Java does not support any form of external dispatch, programmers often compensate by using cascading instanceof tests. This programming pattern is problematic because it is tedious, error-prone, and lacks extensibility [10]. Many instances of this pattern could be re-written to use external methods,

```
let
    widget: Widget(setText: ( ) ⇒ string → unit) = . . .
in
    widget.setText message
```

Figure 4: Code block of Fig. 3a re-written using structural types.

Original Java Code

```
List qlist = ...
Object query = qlist.get(i);
Query q = null;
if (query instanceof String)
   q = parser.parse((String) query);
else if (query instanceof Query)
   q = (Query) query;
else
   System.err.println("Unsupported query type");
```

Unity Re-Write

```
method string.toQuery: ( ) ⇒ QueryParser → Query =
    fun parser: QueryParser --> parser.parse this
method Query.toQuery: ( ) ⇒ QueryParser → Query =
    fun _ --> this

. . .
using toQuery in
    type QueryConvert = Object(toQuery: ( ) ⇒ QueryParser → Query)
    let qlist: List<QueryConvert> = . . .
    let q : Query = qlist.get(i).toQuery(parser)
```

Figure 5: Rewriting instanceof using structural subtyping and external dispatch. At the top is the original code; below is the translated code, which defines the structural type QueryConvert and external methods on Query and String. Note that the translated code eliminates the need for the error condition.

but a problem arises if an instanceof test is performed on an expression of type Object.

To illustrate this, let us consider how instanceof tests would be translated to external methods. Suppose we have a cascaded instanceof, with each case of the form "[else] if expr instanceof C_i { $block_i$ }." This would be translated to an external method f defined on expr's class, and overridden for each C_i by defining $C_i.f$ { $block_i$ }. The bottom part of Fig. 5 shows the external methods translated from the instanceof tests above it (but without an external method defined on Object, the type of query, which we will come to in a moment).

A problem arises when the target expression in the instanceof test is of type Object, as an external method must be defined on Object, then overridden for each type tested via an instanceof. The problem

	instanceof	Expression of type Object	Percentage
Apache collections	225	75	33%
Areca	77	10	13%
JHotDraw	229	50	22%
log4j	54	8	15%
Lucene	56	10	18%
PLT collections	119	64	54%
Smack	56	20	36%
Tomcat	959	158	16%
Average			**26%**

Table 6: Total instanceof tests, the number present in cascading if statements that perform the test on an expression of type Object, and that number expressed as a percentage. Code written using this pattern can be translated to a language with structural subtyping and external dispatch.

with this solution is that it pollutes the interface of Object. In many cases, the implementation of this method performs a generic fallback operation that does not make sense for an object of arbitrary type—but this method becomes part of every class's interface and implementation. (While it is also possible to pollute the interface of an arbitrary class C, this is generally less severe, and detecting such a situation requires application-specific knowledge.)

To determine the prevalence of this pattern, instanceof tests in 8 applications were manually examined. The result of this analysis was that 13% to 54% (with an average of 26%) were performing a cascading instanceof test on an expression of type Object (see Table 6).

Structural subtyping provides one solution to this problem. We have previously defined a language, Unity, that supports both structural subtyping and external dispatch [17]. Using structural types, the type of the expression on which the instanceof is performed would be changed from Object to the structural type consisting of the newly defined external method f. That is, instead of making the target operation applicable to an arbitrary object, it would be applicable to only those objects that contain method f. Figure 5 defines an external method toQuery on String and Query, then uses the structural type { toQuery(...) } as the type for the List elements. The advantage of using structural subtyping is that the main code can call this method uniformly. [6]

Thus, for many applications, there is a potential benefit to using structural subtyping in a language that supports external dispatch; an average of 26% of instanceof tests could be eliminated.

Note that since we refined the element type of the List object, this obviates the need for the error condition—an additional advantage. However, it is not always possible to refine types to a structural type; an expression may simply have type Object, due to the loss of type information. In such a case, it would be possible to re-write the code using a structural downcast. Though the use of casts would not be eliminated, there are still several advantages to this implementation style. First, the external methods could be changed without having to also modify the method that uses them. Also, if subclasses are added, a new internal or external method could be defined for them. Finally, since the proposed cast would use a structural type, it would be more general, applying to any type for which the method were defined.

[6] Note that it would not be possible to make use of a nominal interface containing the method f to call the method in a generic manner. For external methods to be modular, once a method is defined as an internal method, it cannot be implemented with an external method; see [19, 10].

	Uses of getMethod()	Could be rewritten	Percentage
Ant	36	9	25%
Apache Collections	4	3	75%
Areca	1	0	0%
Azureus	27	6	22%
Cayenne	28	4	14%
columba	10	8	80%
DrJava	7	2	29%
emma	2	1	50%
freecol	1	1	100%
hsqldb	2	0	0%
HttpClient	8	6	75%
jedit	10	7	70%
jfreechart	1	1	100%
JHotDraw	26	1	4%
jruby	17	6	35%
jung	1	1	100%
log4j	4	1	25%
openfire	2	0	0%
pmd	2	2	100%
quartz	3	2	67%
struts	2	0	0%
tomcat	37	10	27%
xalan	28	11	39%
Totals	**259**	**82**	**32%**

Table 7: Uses of the reflection method Class.getMethod, and the number and percentage that could be re-written using a structural downcast. Programs that did not call this method are omitted. The percentage entry in the last row is calculated by dividing the total "*could be rewritten*" by the total "*uses of getMethod*."

7 Java Reflection Analysis

We aimed to answer the following question: do Java programs use reflection where structural types would be more appropriate? Uses of reflection likely fall into two categories: cases where dynamic class instantiation and classloading are used, and cases where the type system is not sufficiently powerful to express the programming pattern used. It is difficult to eliminate reflection in the first category, as these uses represent an inherently dynamic operation. However, some of the uses in the second category could potentially be rewritten using structural downcasts. Reducing the uses of reflection is beneficial as it decreases the number of runtime errors and can improve performance.

Across the 29 subject programs, an average of 32% of uses of the reflection method Class.getMethod could be re-written using a structural downcast (see Table 7). A structural downcast is preferable to reflection because type information is retained when later calling methods, as opposed to Method.invoke, which is passed an Object array and must typecheck the arguments at runtime. Additionally, it is easier to combine sets of methods in a downcast; when using reflection, each method must be selected individually. There is also the potential to make method calls more efficient, which is difficult with reflection, due to the low-level nature of the available operations. (For example, the language Whiteoak [14] supports efficient structural downcasts.)

In summary, the high percentage of reflection uses that can be translated to structural downcasts suggests that programmers may sometimes use reflection as a workaround for lack of structural types.

8 Related Work

As mentioned in Sect. 3, researchers have studied the problem of refactoring programs to use most general nominal types where possible [12, 22]. Structural subtyping would make such refactorings more feasible (since new types would not have to be defined) and applicable to more type references in the program (since structural supertypes for library types could be created, while new interfaces cannot).

Muschevici et al. measured the number of cascading instanceof tests in a number of Java programs, to determine how often multiple dispatch might be applicable [20]. They found that cascading instanceof tests were quite common, and that many cases could be rewritten to use multimethods; this is consistent with my results.

Corpus analysis is commonly used in empirical software engineering research. For example, it has been used to examine non-nullness [8], aspects [2], micro-patterns [13], and inheritance [23].

9 Summary and Conclusions

In summary, we found that a number of different aspects of Java programs suggest the potential utility of structural subtyping. While some of the results are not as strong as others, taken together the data suggests that programs could benefit from the addition of structural subtyping, even if they were written in a nominally-typed language. In particular, structural subtyping has the potential be used to improve the reusability and maintainability of existing object-oriented programs.

We presented evidence suggesting that structural subtyping could help make method parameters more general (Sect. 3). There was a high frequency of common methods (Sect 5.1), and a low frequency of common methods representing an accidental name clash (Sect 5.2). We also saw evidence that some cases of code duplication could be avoided with structural subtyping (Sect. 5.3).

Additionally, we showed that existing language designs can lead to coding patterns that defer errors to runtime, coding patterns that could be re-written with structural subtyping to provide more static typechecking in these situations. In particular, some Java runtime exceptions (i.e., OperationUnsupportedException) can be eliminated in a straightforward manner with a design that uses structural subtyping (Sect. 4). Additionally, some uses of Java reflection can be converted to uses of structural subtyping (Sect. 7).

Finally, the study in Sect. 6 showed the synergy between structural subtyping and external dispatch. The data showed that many cases of cascading instanceof tests in Java programs can improved if rewritten using a combination of structural subtyping and external methods, a re-writing which allows an existing class to be adapted to a new context.

We hope that the results of this study will be used to inform designers of future programming languages, as well as serve as a starting point for further empirical studies in this area. Ultimately, one must study the way structural subtyping is eventually used by mainstream programmers; this work serves as a step in that direction.

Acknowledgements

We would like to thank Ewan Tempero for helpful discussions and feedback, and Nels Beckman and the reviewers for comments on an earlier version of this paper. This research was supported in part by the U.S. Department of Defense, Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems," and NSF CAREER award CCF-0546550.

References

[1] R. Amadio and L. Cardelli. Subtyping recursive types. *ACM TOPLAS*, 15(4), 1993.

[2] P. Baldi, C. Lopes, E. Linstead, and S. Bajracharya. A theory of aspects as latent topics. In *OOPSLA*, 2008.

[3] J. Bloch. *Effective Java, Second Edition*. Addison-Wesley, 2008.

[4] J. Boyland and G. Castagna. Parasitic methods: an implementation of multi-methods for Java. In *OOPSLA '97*, pages 66–76, 1997.

[5] G. Bracha and D. Griswold. Strongtalk: typechecking Smalltalk in a production environment. In *OOPSLA '93*, pages 215–230, 1993.

[6] K. Bruce, A. Schuett, R. van Gent, and A. Fiech. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Trans. Program. Lang. Syst.*, 25(2):225–290, 2003.

[7] L. Cardelli. Structural subtyping and the notion of power type. In *POPL '88*, 1988.

[8] P. Chalin and P. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *ECOOP*, 2007.

[9] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP '92*, 1992.

[10] C. Clifton, T. Millstein, G. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM TOPLAS.*, 28(3):517–575, 2006.

[11] K. Fisher and J. Reppy. The design of a class mechanism for Moby. In *PLDI*, 1999.

[12] Florian Forster. Cost and benefit of rigorous decoupling with context-specific interfaces. In *PPPJ '06*, pages 23–30, 2006.

[13] J. Gil and I. Maman. Micro patterns in Java code. In *OOPSLA '05*, pages 97–116, 2005.

[14] J. Gil and I. Maman. Whiteoak: Introducing structural typing into Java. In *OOPSLA*, 2008.

[15] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system, release 3.10. Available at http://caml.inria.fr/pub/docs/manual-ocaml, 2007.

[16] B. Magnusson. Code reuse considered harmful. *Journal of Object-Oriented Programming*, 4(3), November 1991.

[17] D. Malayeri and J. Aldrich. Integrating nominal and structural subtyping. In *ECOOP '08*, July 2008.

[18] Sun Microsystems. Java collections API design FAQ. Available at http://java.sun.com/j2se/1.4.2/docs/guide/collections/designfaq.html, 2003.

[19] T. Millstein and C. Chambers. Modular statically typed multimethods. *Inf. Comput.*, 175(1):76–118, 2002.

[20] R. Muschevici, A. Potanin, E. Tempero, and J. Noble. Multiple dispatch in practice. In *OOPSLA 08*, October 2008.

[21] D. Musser and A. Stepanov. Generic programming. In P. Gianni, editor, *ISAAC '88*, volume 38 of *Lecture Notes in Computer Science*, pages 13–25. Springer, 1989.

[22] F. Steimann. The infer type refactoring and its use for interface-based programming. *Journal of Object Technology*, 6(2), 2007.

[23] E. D. Tempero, J. Noble, and H. Melton. How do Java programs use inheritance? An empirical study of inheritance in Java software. In *ECOOP '08*, pages 667–691, 2008.

A Subjective Criteria

In Section 3.2, we enumerated the number of cases where it could be "useful" to generalize the parameter types of a particular method. To determine this, we asked two questions. First, does the inferred parameter type *S* generalize the abstract operation performed by the method (as determined by the method name)? For example, generalizing the List parameters in ListUtils.intersection *does* appear to generalize the abstract operation of taking the intersection of two sequences. Second, does it seem likely that there would be multiple subtypes of *S*? For example, in Crystal we found that there were two methods of the IBinding interface that were often used, and we were informed by the developers that it was conceivable that they would replace the use of Eclipse binding objects with an application-specific representation.

In Section 5.2, we tabulated the number of methods in a common method group that had "the same meaning." To determine this, we used javadoc when available; when it was not, we examined the body of the method to determine the operation being performed.

Program	Version
Ant	1.7.0
antlr	2.7.6
Apache collections	3.2
Areca	5.5.3
Cayenne	2.0.4
Columba	1.0RC1
Crystal	3.3.0
DrJava	20080904-r4668
Emma	2.0.5312
freecol	0.7.3
hsqldb	1.8.0.4
HttpClient	3.1
jEdit	4.2
JFreeChart	1.0.0-rc1
JHotDraw	7.0.9
jruby	1.0.1
jung	1.7.6
LimeWire	4.13.0
log4j	1.2.15
Lucene	1.4
OpenFire	3.4.2
plt collections	20080904-r4668
pmd	3.3
poi	2.5.1
quartz	1.5.2
Smack	3.0.4
Struts	2.0.11
Tomcat	6.0.14
xalan	2.7.0

Table 8: Version numbers of empirical study subject programs