A Case Study on Type Hints in Method Argument Names in Pharo Smalltalk Projects

Boris Spasojević
University of Bern
Switzerland
Email: spasojev@inf.unibe.ch

Mircea Lungu
University of Groningen
Netherlands
Email: m.f.lungu@rug.nl

Oscar Nierstrasz
University of Bern
Switzerland
Email: oscar@inf.unibe.ch

***Abstract*—A common practice when writing Smalltalk source code is to name method arguments in a way that hints at their expected type (*i.e.*, *aString*, *anInteger*, *aDictionary*). This practice makes code more readable, but the prevalence of this practice is unknown, thus its reliability is questionable. Tools such as the auto complete feature in the Pharo Smalltalk code editor rely on these hints to improve the developer experience. The default algorithm used in Pharo to extract type information from these hints succeeds in extracting a type in slightly over 36% of method arguments taken from 114 Pharo projects. In this paper we present the results of analyzing the failing method argument names, and provide several simple heuristics that can increase the rate of success to slightly over 50%. We also present a case study on the relation between type hints and run-time types of method arguments that shows that type hints, in a great majority of cases, reflect run-time types.**

I. Introduction

Programming languages are usually divided into two groups based on their type system: statically typed languages and dynamically typed languages. Dynamically typed languages are usually considered to be more flexible and more productive [1], but lack the explicit type declarations that typical statically typed languages provide. These explicit type annotations are helpful for program comprehension [2], but can also be used by developer tools (*e.g.*, code completion) to improve the developer experience and productivity.

To partially compensate for the lack of explicit type annotations many Smalltalk developers [3] follow a convention of naming method arguments in a way that hints at the expected type of the method argument [4]. This means that it is recommended to name method arguments by prefixing the expected type with the indefinite article "a" or "an". We call this convention "type hints". Listing 1 presents the implementation of the *indexOf:* method in the *String* class in Pharo Smalltalk [5]. The only argument of this method is named *aCharacter* clearly hinting that the method expects to be called with an object of type Character as the parameter.

This convention obviously helps to provide type information about method arguments in a dynamically typed language, but, as any convention not strictly enforced, is only as good as the discipline of developers to follow it.

In this paper we present a case study of the extent of usage of this convention. We gathered all method arguments from 114 Pharo Smalltalk projects and applied Pharo's built-in system for extracting type information from argument names. We found that this system was able to extract type information from 36.21% of argument names.

```
1 String>>indexOf: aCharacter
2     aCharacter isCharacter ifFalse: [^ 0].
3     ^ self class
4         indexOfAscii: aCharacter asciiValue
5         inString: self
6         startingAt: 1.
```

Listing 1. The implementation of the *indexOf:* method in the *String* class in Pharo Smalltalk. Note that the argument of the method is named *aCharacter* hinting that the expected type is *Character*.

Afterwords we analyzed the argument names that did not yield any type information and, based on the data we observed, developed a few simple heuristics that, when applied, can increase the success rate to 50.69%.

We also note a pattern of expressing so called "Duck-Typed" method arguments *i.e.*, arguments that are expected to be bound to parameters of multiple different types. Almost 1.5% of all method arguments are named in this manner, so any tool attempting to extract type information from method arguments should be aware of this pattern.

To explore whether type hints actually reflect run-time types of arguments we conducted a small study by collecting run-time type information for arguments of two projects and comparing them to their type hints. We find that, on average 76% of type hints reflect run-time types. We discuss the misleading arguments, most of which would be understood by a developer with domain knowledge, and classify most of them into several patterns.

This paper is organized as follows: section II describes the process of gathering the argument names and Pharo's built-in system for extracting type information from argument names; section III discusses the duck-typed method arguments and explains that they are treated as a special case; section IV discusses the proposed heuristics to improve the success rate; section V gives a final overview of the data and conclusions in the previous sections; The study of the correlation between type hints are run-time types is shown in section VI; section VII and section VIII discuss related and future work respectively and finally section IX concludes.

II. Data acquisition

To evaluate the scope of usage of type hints we first gather a large set of method arguments from open source Pharo Smalltalk projects. For our data source we chose 114 projects defined in the "Configuration Browser". The configuration browser is a tool to automatically load Smalltalk project source code, similar to Maven[1] for Java. At the time of data acquisition[2] the configuration browser defined 145 projects, but 31 projects failed to load so they where removed from the data set.

From these 114 projects we extracted a total of 146,297 arguments. We call this set of all arguments *Arg*. To proceed further in our analysis we first need to understand how the process of extracting type information from argument names in Pharo works.

A. The Type Guesser Built into Pharo

The default tool used for extracting type information from argument names in Pharo is part of the code completion tool. The code completion tool is called "NEC" and is based on the eCompletion[3] package developed by Ruben Baker. The process of extracting type information from argument names is refereed to as "Type Guessing" and is encapsulated in the class side method *getClassFromTypeSuggestingName:* of class *NECVarTypeGuesser*[4].

The implementation of this method is quite simple and is presented in graphical notation in Figure 1. The edge labels represent the data flow for two example input strings (argument names) *i.e.*, "*aCharacter*" and "*anInteger*". This method first removes the leading character of the input argument, and attempts to find and return a class with that name in the system. Failing that, the method removes all characters before the first capital letter of the input arguments, and repeats the attempt. Failing both times, the methods returns *nil*[5].

B. Initial Results

Using the type guesser in Pharo we divided the *Arg* set into two subsets: *Succ* (Successfully guessed) - those from which a type was successfully extracted; and *Fail* (Guess failed) - those that did not yield any type. The definition of these sets is given in Figure 2. The function *guessType* is an abstraction of the method *getClassFromTypeSuggestingName*, and returns a set of possible types.

The results of applying *NECVarTypeGuesser* to our data set are shown in Table I. We can see that *NECVarTypeGuesser* guessed the types of fewer than 37% of all arguments. These arguments contain clear type hints and thus are of no further interest to us for further analysis. We continue only on the 63.79% of method arguments that failed to yield a type (the *Fail* set), in an attempt to understand why this is and to improve the success rate.

[1]http://maven.apache.org

[2]Date: 09.03.2015.

[3]http://uncomplex.net/ecompletion/

[4]This class is part of the base Pharo image which can be obtained at http://get.pharo.org

[5]*nil* is the Smalltalk equivalent of *null* in Java or *nullptr* in C++

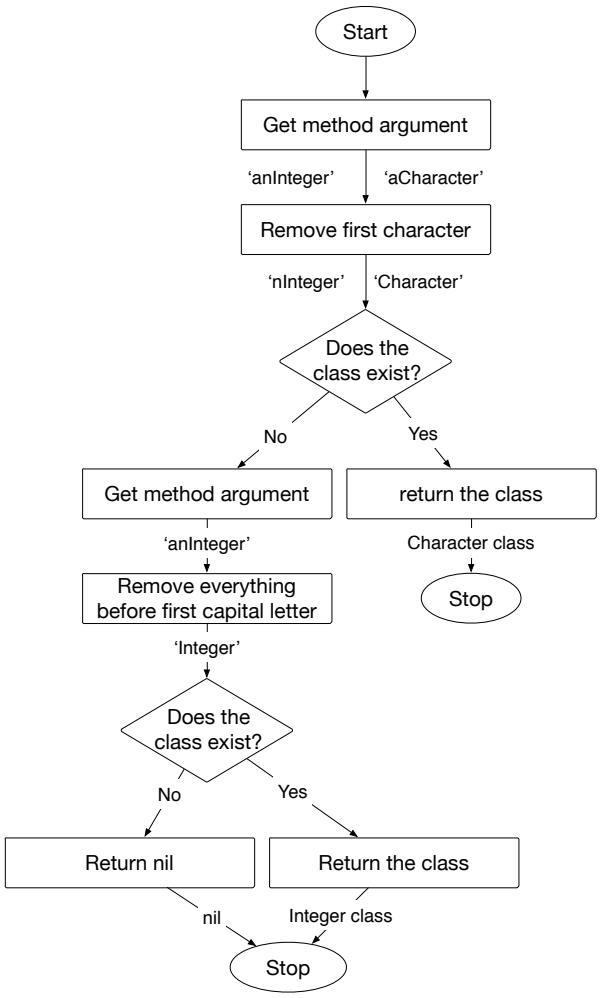

Fig. 1. An activity diagram of the implementation of *NECVarTypeGuesser>>getClassFromTypeSuggestingName*. Edge labels represent data flow for example inputs *'aCharacter'* and *'anInteger'*

$$guessType : Arg \rightarrow \{Type\} \tag{1}$$

$$Succ = \{a | a \in Arg, guessType(a) \neq \emptyset\} \tag{2}$$

$$Fail = \{a | a \in Arg, guessType(a) = \emptyset\} \tag{3}$$

Fig. 2. The core sets. *Arg* = all arguments, *Succ* = type guessed, *Fail* = type not guessed.

TABLE I
THE NUMBER AND PERCENTAGE OF METHOD ARGUMENTS WHICH DO AND DO NOT PRODUCE A TYPE USING *NECVarTypeGuesser*.

	#	% of $\|Args\|$
$\|Arg\|$	146,297	100%
$\|Succ\|$	52,981	36.21%
$\|Fail\|$	93,316	63.79%

III. Duck-Typed Method Arguments

After starting the manual inspection of the method arguments for which *NECVarTypeGuesser* failed to guess a type, we noticed that a substantial number of argument names refer to more than one type. This kind of method arguments are usually referred to as "Duck-Typed"[6] [6]. The term comes from the duck test, attributed to James Whitcomb Riley: *"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck"*. In the context of method arguments, this means that the method does not expect a parameter of a particular type, but rather of any type that follows a certain interface. For the sake of simplicity, we consider any method argument that specifies multiple possible types to be duck-typed.

We find that the pattern for expressing that an argument can take on multiple potential types is to concatenate all possible types with the word "Or". For example, one of the most common argument names with this property is *aStringOrByteArray* appearing 99 times in our corpus. To a developer this is a clear message that this argument should be either of type *String* or *ByteArray*, but the implementation of *NECVarTypeGuesser* does not consider this pattern and unsuccessfully attempts to find a class called *StringOrByteArray*.

A. Impact of Duck-Typed Arguments

To calculate the impact of duck-typed arguments, we extracted all argument names that match the following regular expression.

$$.*Or[A-Z].* \tag{4}$$

The word "Or" should be followed by a capital letter to ensure that, due to Camel Notation [7], we only match that word and not words like "Original", "Ordered" *etc.* A total of 2139 method arguments matched this regular expression.

Manual inspection revealed that 36 occurrences of the extracted arguments do not in fact refer to multiple types. These are all occurrences of two different argument names: *aBlockWithZeroOrOneParameter* and *aZeroOrOneArgBlock*. It is obvious that these argument names refer to a varying number of arguments of a *BlockClosure i.e.*, Lambda expression, and not multiple possible types.

So the very simple regular expression based heuristic we used thus far can easily be integrated in any tool, and on our data set has a false positive rate of only 1.68%. We consider all arguments whose name match the regular expression from Equation 4 but are not duck-typed to be false positives *i.e.*, 36 occurrences of *aBlockWithZeroOrOneParameter* and *aZeroOrOneArgBlock*. A more complex heuristic based on natural language processing could also be an option.

With this in mind, we define the set of duck-typed method arguments (*Duck*) in Figure 3 as the set that matches our regular expression from Equation 4 excluding *aBlockWithZeroOrOneParameter* and *aZeroOrOneArgBlock*. The *name* function extracts the name of the argument as a string.

$$\begin{aligned} \mathit{Duck} = \{a | a \in \mathit{Fail},\ & \\ & \mathit{regexMatch}(a, ".*Or.*[A-Z]"), \\ & \mathit{name}(a) \neq "\mathit{aBlockWithZeroOrOneParameter}", \\ & \mathit{name}(a) \neq "\mathit{aZeroOrOneArgBlock}"\} \end{aligned} \tag{5}$$

Fig. 3. The *Duck* set contains all arguments that hint at multiple types.

The *Duck* set contains a total of 2,103 method arguments or 1.44% of the *Arg* set. This is not an insignificant percentage and any tool attempting to guess types should be aware of the existence of this pattern.

B. Distribution of Number of Types in Duck-Typed Arguments

A followup question regarding duck-typed method arguments is how many different types are hinted at in these arguments. The distribution is presented in Table II. As per intuition, the vast majority (94.82%) of duck-typed arguments hint at two types (*e.g.*, *aStringOrByteArray*, *aUrlOrString*, *aStringOrText*). Around 7% hint at three types (*e.g.*, *aStringOrCollectionOrBlock*, *aDateOrNumberOrString*), and less than half of a percent hint at more. The maximum number of different hinted types is 5, with the argument name *aSelectorOrElementOrjQueryOrBooleanOrNumber* [*sic*]. This might be considered an unwieldy argument name, but on the other hand, it completely defines the number of expected types within the source code with no need for additional documentation.

TABLE II
Distribution of the number of different types in duck-typed method arguments.

Different hinted types	Number of occurrences	% (of $\|Arg\|$)	% (of $\|Duck\|$)
5	5	0.00%	0.24%
4	5	0.00%	0.24%
3	149	0.10%	7.09%
2	1994	1.36%	94.82%

IV. Heuristics for Type Hints

We continue the manual inspection of arguments that failed to yield a type is on the set *Fnd* defined in Figure 4. This is a set of all arguments from the *Fail* set that are not duck-typed (*Fnd* — failed, non duck-typed). It contains 91,213 arguments which is 62.35% of the *Arg* set or 97.75% of the *Fail* set. The aim of further inspection is to identify subsets of *Fnd* that contain type hints, and identify heuristics for identifying the types.

[6]We acknowledge that all arguments in Smalltalk are potentially "Duck-Typed". We use this term to note the user-specified occurrence of an argument being potentially bound to different types at run time.

$$Fnd = Fail \backslash Duck \tag{6}$$

Fig. 4. The *Fnd* set contains all arguments from the *Fail* set that are not duck-typed.

A. spec *and* html

In the *Fnd* set we find a frequent occurrence of the arguments *spec* and *html*.

Inspection of the source code reveals that *spec* is the standard name used for specifications of Metcello versions. Metacello is a package management system for Monticello, a distributed version control system for Smalltalk. The implementation details are not important, but we can claim that the arguments named spec are of type *MetacelloAbstractVersionConstructor*, as it is the superclass for all classes used to construct Metacello versions. So we define the *Spec* set in Figure 5 Equation 7. This set contains 6,132 elements or 4.19% of the *Arg* set.

A similar situation exists with the arguments named *html*. A common practice when writing applications using Seaside [8], a web development framework for Smalltalk, is to pass the object representation of the HTML element as an argument to methods of objects that perform an action on it (usually the object renders itself on the HTML element). All of these arguments are instances of *WAHtmlCanvas*. As with the *spec* argument name we define the *Html* set in Figure 5 Equation 8, which contains 2,935 elements or 2.01% of the *Arg* set.

$$Spec = \{a | a \in Fnd, name(a) = \text{“}spec\text{”}\} \tag{7}$$

$$Html = \{a | a \in Fnd, name(a) = \text{“}html\text{”}\} \tag{8}$$

Fig. 5. The *Spec* and *Html* sets contain arguments from the *Fnd* set that are named *spec* and *html* respectively

B. Blocks, Strings and Collections

Using block closures in Smalltalk is common practice. The class *BlockClosure* offers the default implementation. Unfortunately for the default type guessing algorithm, a majority of method arguments that are expected to be an instance of *BlockClosure* are not named *aBlockClosure* (only 47 occurrences of arguments named *aBlockClosure* in *Arg*) but rather *aBlock* (6,167 occurrences of arguments named *aBlock* in *Arg*) as Figure 6 summarizes. Using *aBlock* rather than *aBlockClosure* is even present in the book “Smalltalk Best Practice Patterns” [4] by Kent Beck.

To give more context to arguments hinting at *BlockClosure*, developers often add additional descriptors to the argument name. Examples of such argument names are *toBlock*, *fromBlock*, *anErrorBlock*, *aOneArgBlock*, *aFormatBlock etc.* Also, a substantial number of arguments are named simply *block*, ignoring the article.

In order to group all these different ways of specifying arguments of type *BlockClosure* we define a set called *Block′*

$$|\{a | a \in Arg, name(a) = \text{“}aBlockClosure\text{”}\}| = 47 \tag{9}$$

$$Block = \{a | a \in Arg, name(a) = \text{“}aBlock\text{”}\} \tag{10}$$

$$|Block| = 6,167 \tag{11}$$

Fig. 6. Many more arguments are named *aBlock* than *aBlockClosure*

as the set of all arguments whose name matches the regular expression “. $*$ $(B|b)lock.*$”. This is formally defined in Figure 7 Equation 12.

Following the same logic we define two more sets. The first attempts to group all arguments hinting at the *Collection* type - *Coll* (Figure 6 Equation 13) and the second for arguments ginting at the *String* type - *String* (Figure 6 Equation 14).

The number of elements in all of these sets can be found in Table III. It is worth noting that the set *Block′* contains over a thousand more elements than set *Block*, showing that using the simple heuristic can attach a type to a much larger set of arguments.

TABLE III
THE CARDINALITY OF THE *Block′*, *Coll* AND *String* SETS.

	#	% (|*Arg*|)
|*Block′*|	7,886	5.39%
|*Coll*|	559	0.38%
|*String*|	1,793	1.23%

C. Duplicate entries in sets Block′, Coll and String

The regular expression based definitions of sets *Block′*, *Coll* and *String* are quite naive, and further inspection of the elements of these sets reveals that certain arguments appear in multiple sets as shown in Figure 7 Equation 17. The problem arises in argument names that match multiple regular expressions.

The number of such elements is fairly small. A total of 18 arguments appear in the *String* and *Coll* sets, and all are named *aCollectionOfStrings*. Their name is a clearly hinting at the type *Collection* rather than *String*. Similarly, 12 arguments appear in the *Block′* and *String* sets, and are all named *aBlockAnsweringAString*, hinting at the type *BlockClosure* and not *String*. No arguments are present in both *Block′* and *Coll* sets.

With this in mind we can conclude that, in the *Fnd* set, a clear rule can be observed for dealing with these ambiguities. We notice that all arguments that appear in these three sets can be placed in the adequate set by following a strict hierarchy of set priorities: blocks are higher priority than collections which are higher priority than strings.

Namely, all duplicate arguments from the *Coll* set, are properly placed, since the duplicates named *aCollectionOfStrings* are clearly collections, thus collections have a higher priority than strings. Similarly, all duplicate arguments from

$$\textit{Block}' = \{a | a \in \textit{Fnd}, \textit{regexMatch}(a, \text{“}.*(b|B)lock.*\text{”})\} \quad (12)$$

$$\textit{Coll} = \{a | a \in \textit{Fnd}, \textit{regexMatch}(a, \text{“}.*(c|C)ollection.*\text{”})\} \quad (13)$$

$$\textit{String} = \{a | a \in \textit{Fnd}, \textit{regexMatch}(a, \text{“}.*(s|S)tring.*\text{”})\} \quad (14)$$

$$\textit{Coll}' = \{a | a \in \textit{Fnd}, \textit{regexMatch}(a, \text{“}.*(c|C)ollection.*\text{”}), \textit{notRegexMatch}(a, \text{“}.*(b|B)lock.*\text{”})\} \quad (15)$$

$$\textit{String}' = \{a | a \in \textit{Fnd}, \textit{regexMatch}(a, \text{“}.*(s|S)tring.*\text{”}), \textit{notRegexMatch}(a, \text{“}.*(b|B)lock.*\text{”}), \textit{notRegexMatch}(a, \text{“}.*(c|C)ollection.*\text{”})\} \quad (16)$$

$$\begin{aligned} \textit{Block}' \cap \textit{Coll} &= \emptyset \\ |\textit{Block}' \cap \textit{String}| &= 12 \\ |\textit{String} \cap \textit{Coll}| &= 18 \end{aligned} \quad (17)$$

$$\begin{aligned} \textit{Block}' \cap \textit{Coll}' &= \emptyset \\ \textit{Block}' \cap \textit{String}' &= \emptyset \\ \textit{String}' \cap \textit{Coll} &= \emptyset \end{aligned} \quad (18)$$

Fig. 7. Sets of arguments based on regular expressions

the *Block′* set, are properly placed, since the duplicates named *aBlockAnsweringAString* are clearly blocks. Thus blocks have a higher priority than strings. We artificially introduce the rule that blocks are higher priority than collections in order to make our heuristic complete. This rule might cause false positives in the cases such as the hypothetical argument name *aCollectionOfBlocks*, placing such an argument in the set of blocks rather than in the set of collections where it would belong.

Another approach to removing duplicate entries requires a more thorough analysis of these cases, either by natural language processing techniques or by focusing on splitting the argument name by “Of” and determining the priorities by the order. We feel this would introduce a lot of complexity for not much gain and leave out such attempts.

We apply these hierarchy rules in defining the sets *Coll′* (Figure 7 Equation 15) and *String′* (Figure 7 Equation 16), and ensure that there is no overlap between these sets (Figure 7 Equation 18). The function *notRegexMatch* is, as the name hints, a negation of the *regexMatch* function.

TABLE IV
THE CARDINALITY OF THE *DuckS* AND *DuckF* SETS.

	#	% (|*Arg*|)
|*DuckS*|	1,905	1.30%
|*DuckF*|	198	0.14%

$$\textit{DuckF} = \textit{Duck} \backslash \textit{DuckS} \quad (19)$$

Fig. 8. The *DuckF* set contains all duck-typed arguments whose type could not be guessed.

D. Guessing types of duck-typed arguments

All the sets defined in this section thus far are subsets of *Fnd*, meaning that the arguments we defined as duck-typed are not included in any of the sets. To determine the set of duck-typed arguments whose type can be guessed (*DuckS* $\subset$ *Duck*) we follow a simple algorithm:

1) Split the name of the argument $a \in Duck$ by the keyword *Or*
2) Treat each of the sub-names as a valid name of a hypothetical argument b
3) If $\textit{guessType}(b) \neq \emptyset$ than we guessed the type of a, and $a \in$ *DuckS*.
4) If not, assume that $b \in$ *Fnd*
5) If it holds that $b \in \textit{Spec} \vee b \in \textit{Html} \vee b \in \textit{Block}' \vee b \in \textit{Coll}' \vee b \in \textit{String}'$ then we guessed the type of a, and $a \in$ *DuckS*.
6) If not, $a \notin$ *DuckS*

Essentially, if we can guess one of the types that the argument name hints at, we declare the argument type guessed. So the set *DuckS* holds all arguments from the set *Duck* whose type can be guessed. The remaining arguments make up the *DuckF* set defined in Figure 8. The cardinality of these sets is given in Table IV.

V. FINAL RESULTS

With all the heuristic based sets defined in section IV we have exhausted the ways in which we can guess types in the *Arg* set. The potential for other heuristics still exists, *i.e.*, arguments named *index* can be considered to be integers, plural nouns can be considered collections *etc.* but without additional studies dedicated to these situations we cannot claim that these potential heuristics are well-reasoned.

The set H, defined in Figure 9 Equation 20, is the union of all sets defined by heuristics and accounts for 13.18% of all arguments. Now, we can finally define a set of all arguments whose type can be guessed from the name. We call this set *Succ′* and define it in Figure 9 Equation 21. Also, in Figure 9 Equation 23, we define the set *Fail′*, as the set of all arguments whose types can not be guessed from the name. It is defined as the union of all duck-typed arguments whose type is not

TABLE V
THE TOP TEN MOST FREQUENT ARGUMENT NAMES IN THE F SET.

Argument name	#	% ($\|Arg\|$)
n	1511	1.03%
aName	1440	0.98%
a	1118	0.76%
lda	1092	0.75%
nodes	868	0.59%
aBrick	825	0.56%
work	816	0.56%
aValue	786	0.54%
info	753	0.51%
evt	670	0.46%

TABLE VI
THE CARDINALITY OF ALL DEFINED SETS. HIERARCHY REPRESENTS SUBSET RELATION.

				#	% $\|Args\|$
$\|Arg\|$				146,297	100%
$\|\longrightarrow$	$\|Succ'\|$			74,161	50.69%
$\|$	$\|\longrightarrow$	$\|Succ\|$		52,981	36.21%
$\|$	$\|\longrightarrow$	$\|DuckS\|$		1,905	1.30%
$\|$	$\|\longrightarrow$	$\|H\|$		19,275	13.18%
$\|$		$\|\longrightarrow$	$\|Block'\|$	7,886	5.39%
$\|$		$\|\longrightarrow$	$\|String'\|$	1,793	1.92%
$\|$		$\|\longrightarrow$	$\|Coll'\|$	559	0.38%
$\|$		$\|\longrightarrow$	$\|Spec\|$	6,132	4.19%
$\|$		$\|\longrightarrow$	$\|Html\|$	2,935	2.01%
$\|\longrightarrow$	$\|Fail'\|$			71938	49.31%
	$\|\longrightarrow$	$\|DuckF\|$		198	0.14%
	$\|\longrightarrow$	$\|F\|$		71,938	49.17%

guessable (*DuckF*) and all non-duck-typed arguments whose type is not guessable (F).

The cardinality of these sets, as well as all their subsets is presented in Table VI. We can see that the default type guessing implementation can be substantially improved by incorporating the proposed heuristics. The total number of arguments whose type can be guessed is 74,161 or 50.69% of all arguments. This set is by no means complete. If we look at just the top ten most frequent names of arguments from the F set show in Table V, we can see that there are still argument names that contain hints *i.e.*, *aName*, *aBrick* and *aValue*. These hints are more delicate and might carry a lot more meaning for a developer with domain knowledge, but providing tool support for such cases is a more challenging task. With all this in mind, we can thus conclude that type hints are a commonly used way to name method arguments in Smalltalk projects, and that even fairly simple tool support can work about 50% of the time.

VI. QUALITY OF TYPE HINTS

So far we have focused on the quantity of type hints in a large number of Smalltalk projects. In this section we conduct a separate analysis on the quality of type hints in two Smalltalk projects: Glamour [9] , a framework for describing navigation flow of GUI data browsers; and Roassal [10], a visualization

$$H = Spec \cup Html \cup Block' \cup Coll' \cup String' \quad (20)$$

$$Succ' = Succ \cup H \cup DuckS \quad (21)$$

$$F = Fnd \backslash H \quad (22)$$

$$Fail' = F \cup DuckF \quad (23)$$

Fig. 9. The H set contains all arguments all sets defined by heuristics in section IV, the *Succ*'set contains all arguments whose type is guessable and *Fail*$'$ contains all arguments whose type is not guessable.

engine for Smalltalk. The reason we chose these two projects for our analysis is their rich example library which enables us to easily run a dynamic analysis similar to real-world usage of these projects. The end goal of this analysis is to verify whether types extracted from type hints match run-time types of arguments.

A. Acquisition of run-time types

In order to collect run-time types of method arguments in our case projects we instrumented the source code of these projects using a slightly modified version of a tool called "Variable Tracker"[7] used for gathering run-time type information from Smalltalk projects. The modification was to limit the tool to method arguments only.

After instrumenting the source code, we executed the examples for each project. Glamour defines 68 examples of which one failed to execute. Roassal defines 63 examples of which four failed to execute. This is by no means an exhaustive dynamic analysis, but it does provide a usage similar to real world applications of these frameworks.

The result of this is presented in the "Total arguments" entry of Table VII. As the table shows, we collected run-time type information on 251 and 559 arguments from Glamour and Roassal respectively.

During normal execution some of these arguments might get multiple different types due to Smalltalks dynamic type system. We did not encounter such situations during our dynamic analysis. A more thorough dynamic analysis might yield such cases.

TABLE VII
A SUMMARY OF THE DYNAMIC ANALYSIS RESULTS.

	Glamour	Roassal
Total arguments	251	559
Contain type hint	141	159
Good type hints	126	103
Misleading type hints	15	56

B. Type hints and run-time types

To assess the quality of type hints we first extract the arguments that contain type hints. We include all arguments that contain type hints according to the type guesser described

[7]http://smalltalkhub.com/#!/~rostebler/VariableTracker

in subsection II-A as well as all those that match the heuristics described in section IV. The result is presented in the "Contain type hint" entry in Table VII. A total of 141 and 160 arguments contain type hints in Glamour and Roassal.

Finally, we separate the arguments whose names contain type hints that match the run-time type or one of its superclasses. We include the superclasses of the run-time type in order to include occurrences of subtype polymorphism [11]. A typical toy example would be an argument named *anAbstractShape* getting the type *"Circle"* (a subclass of *AbstractShape*) at run time. The results of this separation is shown in the "Good type hints" entries in Table VII. We can see that in Glamour 89.36% of type hints are good, meaning the guessed type or one of its subtypes is used at run time. In Roassal this number is much lower at 64.77% and the average across both applications is 76%.

C. Misleading type hints

The last entry in Table VII ("Misleading type hints") shows the number of arguments whose type hint did not match the run-time type. There is a total of 71 such arguments, 15 from Glamour and 57 from Roassal. Many of these arguments are misleading to our tools, but might not be to a developer with more context and domain knowledge. An overview of all these arguments is presented in Table VIII.

We can notice a few distinct patterns in this data, and their description follows. All but 11 arguments (15.49%) fall into one of these patterns.

1) Class clash: This situation emerges when the guessed type exists, but the run-time type has a similar name and no hierarchical connection. Examples of this are

- Arguments that hint at the type *Browser* (a class present in the default Pharo image) but at run time receive *GMLBrowser* (GLM stands for Glamour)
- Arguments that hint at the type *Shape* or *Canvas* (both classes present in the default Pharo image) but at run time receive some subclass of *TRShape* or *TRCanvas* (TR stands for Trachel, a module used in Roassal) *etc.*

This pattern accounts for 36.62% (26 out of 71) arguments with misleading type hints.

2) Any Object data model: Roassal is a very flexible visualization engine and can visualize any set of objects and their connections. To make this possible, it relies on using any object as a data model for generating the visualization, and specifying through dynamic features of Smalltalk *i.e.*, metaprogramming, how to interact with the model. That is why it is common to find arguments named *aModel* that get bound to many different types at run time. The problem comes due to the fact that *Model* is a class in the default Pharo image. In our set of misleading type hints this pattern accounts for 8.45% (6 out of 71) arguments.

3) Block or Value: In 18.3% (13 out of 71) cases of misleading types we notice a duck-typed argument that starts with *aValue* and also hints at expecting a *BlockClosure*. A "Value" in this case can be any object, and in case a *BlockClosure* is provided, it is assumed that evaluating it will produce the expected value. A more exhaustive dynamic analysis might produce cases where this argument is bound to an instance of *BlockClosure*. This pattern is very similar to "any object as a data model" but we separate them because *Model* is a class in the Pharo image, and *Value* is not. The type hint in this pattern comes from the hinted option of using instances of *BlockClosure*.

The reason why Roassal can expect any object as a value is that it extends the class *Object*[8] with the method *rtValue:* which is the only method invoked on arguments from this pattern. The default implementation of this method just returns the receiver object, but it is overridden in the *BlockClosure* class to evaluate the closure with the method arguments.

4) Block or Symbol: A common idiom in Smalltalk source code is that methods that expect an instance of *BlockClosure* can also accept an instance of *Symbol* [5].

This is illustrated well by the filtering method *select:* of the collections package in Smalltalk. This method expects an instance of *BlockClosure* that describes the criteria for selection of elements of the collection. If this criteria is to only invoke a single method with no arguments (for example, the *isZero* method of class *Number*), than we can provide just the selector (method name) as an instance of *Symbol*. The result is smaller code that is more readable. This idiom exploits duck-typing by implementing the *value:* method — normally associated with block closures — for the *Symbol* class in the obvious way.

This pattern accounts for 18.3% (13 out of 71) of the misleading type hints. As with the "Value or Block" pattern, a more thorough dynamic analysis would most likely find cases where these arguments would be bound to instances of *BlockClosure*.

5) Convertible: We found only two occurrences of this pattern, but it is a situation that can theoretically happen much more often. Essentially, the argument was hinting at expecting an instance of the class *Float*, but received an instance of the class *SmallInteger* at run time. The *SmallInteger* class is not a specialization of the *Float* class, but implicitly, because $\mathbb{N} \subset \mathbb{R}$, any integer is also a float.

The usage of this argument is restricted to arithmetic operations making the instance of *SmallInteger* completely indistinguishable from an instance of *Float*. This is because all the arithmetic operations in Pharo are developed in a way that provides implicit conversion to the appropriate type when necessary *i.e.*, before invoking the VM primitive that performs the calculation.

VII. Related work

To the best of our knowledge this is the only conducted study of type hints. The closest body of related work consists of type inference approaches for dynamically typed languages. Such type inference is similar to type guessing as both aim to increase program comprehension and tool support. This

[8] Just like in Java, all classes in Smalltalk eventually inherit from the *Object* class

TABLE VIII
OVERVIEW OF ALL ARGUMENT NAMES THAT HAVE MISLEADING TYPE HINTS.

Project	Argument name	Run time class	Pattern	#
Glamour	aBrowser	GMLBrowser	Class clash	12
	anAnnouncement	Announcement class	-	2
	aSymbolOrABlock	ByteString	-	1
Roassal	aModel	ByteString, Association, SmallInteger, *etc.*	Any object data model	6
	toBlock, fromBlock, followBlock	ByteSymbol	Block or symbol	13
	aValueOrOneArgBlock	Color, SmallInteger, ByteSymbol, *etc.*	Block or value	11
	aValueOrASymbolOrOneArgBlock	SmallInteger	Block or value	2
	trachelShape, anotherShape, aShape	some subclass of TRShape	Class clash	11
	trachelCanvas	TRCanvas	Class clash	2
	aBehaviour	TRNoBehaviour	Class clash	1
	aFloat	SmallInteger	Convertable	2
	aKey	ByteSymbol	-	2
	aBlock	SmallInteger	-	1
	aLineShapeOrBlock	TRLine class	-	1
	aTRLine	TRBezierLine	-	1
	aTRShape	TRLine class	-	1
	depClass	String class	-	1
	listOfElementsOrOneArgBlock	TRGroup	-	1

is known to be used for better comprehension of COBOL programs [12], [13], [14], [15]. The main insight in such work is that comparing variables using relational operators shows that their types must be compatible, and that if an expression is assigned to a variable, the type of the expression must be a subtype of that variable.

Guo *et al.* developed an analysis for x86 compiled binaries and Java bytecode that infers abstract types, and conducted a user study showing that having type information can help with reverse engineering problems. Their main insight was, similarly to the COBOL community, to track interactions between variables and infer abstract types. Their approach is based on run-time information. A similar goal was achieved using static analysis by Robert OCallahan and Daniel Jackson with their Lackwit tool [16] for C.

In our previous work we used heuristics based on ecosystem data to improve an existing type inference engine for Smalltalk [17]. By observing the method–type relations in the ecosystem and extending an existing type inference engine with this knowledge we achieved a 100% improvement on our evaluation.

A different motivation for type inference techniques in statically typed languages like Scala [18], or ML [19] is to free the developer from having to specify types that can be inferred, or identifying run-time types at compile type in the presents of subtype polymorphism [20].

Many different type inference techniques are developed with the aim to impose a more strictly controlled type system, and enable certain type checks at compile time. This is a different goal compared to type hints, which are intended primarily for program comprehension. Many such attempts for Smalltalk [21], [22], [23] and other dynamically typed languages such as Python [24] and Ruby [25] have been made.

VIII. FUTURE WORK

We envision three directions of future work: continuously monitoring the way arguments in given projects change over time (subsection VIII-A); attempting to integrate dynamic analysis to infer patterns between argument names and their run-time types (subsection VIII-B); and performing a comparison to type inference engines (subsection VIII-C).

A. Continuous monitoring

Code evolves, and with time new patterns in type hints might appear. A heuristic based type guessing tool would be only as good as its heuristics, so continuously monitoring argument names from projects of interest could identify emerging trends in argument names, and notify the tool developer of the new patterns. This would prompt the developer to investigate the new patterns, develop an appropriate heuristic and integrate it in the tool.

B. Dynamic analysis

An alternative to observing static changes in source code could be observing a system in question at run time, and attempting to find patterns of mappings between argument names and run-time types. This could be done automatically through machine learning techniques, and could be tailored to the project domain helping with some of the bad type hints from section VI *i.e.*, an argument named *aShape* can hint at the *Shape* class or *TRShape* class, and run-time information can distinguish which.

C. Comparison with type inference engines

The standard approach for statically obtaining type information in dynamically typed languages is to perform type inference. Many different type inference approaches exist [26], [27] and some approaches have existing implementations in Pharo [17], [28]. The question is whether type hints or type inference can provide more type information for method

arguments in Smalltalk, or whether their combination is worth the effort.

IX. Conclusion

In this paper we present two case studies, one on how frequently type hints are used in method argument names in Smalltalk and one on the quality of those type hints in relation to run-time types.

In the first study we analyze a total of 146,297 arguments taken from 114 Pharo Smalltalk projects, and conclude that the existing tool for type guessing has a 36.21% success rate. We manually analyze the arguments that failed to yield a type, and propose several heuristics that improve the percentage of guessed types to 50.69%. This percentage is not final, as many other heuristics potentially exist, but further inquiry and more domain knowledge is required to formulate them. So the main conclusion of the study is that at least one in two method arguments in Smalltalk projects contains a useful type hint. Another conclusion drawn from this case study is that 1.44% of method arguments hint at multiple types We present a very simple heuristic for identifying such method arguments with a false positive rate of only 1.68% in our data set.

In the second study we collected run-time types of method arguments from two projects with a rich set of examples used as input, and compared the run-time types to the types extracted from type hints. We find that in 76% of cases the type hint matches the run-time type when controlling for subtype polymorphism. We also present an analysis of the misleading type hints, and identify several patterns that better our understanding of why the type hints are misleading.

We propose several directions of future work, most on improving Smalltalk tools for type guessing using the information from the conducted studies. Beside that, replicating these studies with different languages would help broaden and generalize the understanding of type hints.

Acknowledgment

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project No. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015). We also gratefully acknowledge the financial support of the Swiss Group for Object Oriented Systems and Environments (CHOOSE – http://choose.s-i.ch/) and the European Smalltalk User Group (ESUG – http://www.esug.org/).

References

[1] S. Hanenberg, "An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time," *SIGPLAN Not.*, vol. 45, no. 10, pp. 22–35, Oct. 2010. [Online]. Available: http://doi.acm.org/10.1145/1932682.1869462

[2] C. Mayer, S. Hanenberg, R. Robbes, E. Tanter, and A. Stefik, "An empirical study of the influence of static type systems on the usability of undocumented software," *SIGPLAN Not.*, vol. 47, no. 10, pp. 683–702, Oct. 2012. [Online]. Available: http://doi.acm.org/10.1145/2398857.2384666

[3] A. Goldberg and D. Robson, *Smalltalk 80: the Language and its Implementation*. Reading, Mass.: Addison Wesley, May 1983. [Online]. Available: http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf

[4] K. Beck, *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997. [Online]. Available: http://stephane.ducasse.free.fr/FreeBooks/BestSmalltalkPractices/Draft-Smalltalk%20Best%20Practice%20Patterns%20Kent%20Beck.pdf

[5] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker, *Pharo by Example*. Square Bracket Associates, 2009. [Online]. Available: http://pharobyexample.org

[6] D. Thomas, C. Fowler, and A. Hunt, *Programming Ruby 1.9: The Pragmatic Programmers' Guide*, 3rd ed. Pragmatic Bookshelf, 2009.

[7] A. Wiese, V. Ho, and E. Hill, "A comparison of stemmers on source code identifiers for software search," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, Sep. 2011, pp. 496–499.

[8] S. Ducasse, A. Lienhard, and L. Renggli, "Seaside: A flexible environment for building dynamic web applications," *IEEE Software*, vol. 24, no. 5, pp. 56–63, 2007. [Online]. Available: http://scg.unibe.ch/archive/papers/Duca07a-SeasideIEEE-SCG.pdf

[9] P. Bunge, T. Gîrba, L. Renggli, J. Ressia, and D. Röthlisberger, "Scripting browsers with Glamour," European Smalltalk User Group 2009 Technology Innovation Awards, Aug. 2009, glamour was awarded the 3rd prize. [Online]. Available: http://scg.unibe.ch/archive/reports/Bung09bGlamour.pdf

[10] V. P. Araya, A. Bergel, D. Cassou, S. Ducasse, and J. Laval, "Agile visualization with Roassal," in *Deep Into Pharo*. Square Bracket Associates, Sep. 2013, pp. 209–239.

[11] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, pp. 471–522, Dec. 1985. [Online]. Available: http://lucacardelli.namehttp://lucacardelli.name/Papers/OnUnderstanding.A4.pdf

[12] A. Deursen and L. Moonen, "Type inference for COBOL systems," in *Proceedings of WCRE '98*. IEEE Computer Society, 1998, pp. 220–229, iSBN: 0-8186-89-67-6.

[13] A. Deursen and L. M. Moonen, "Understanding cobol systems using inferred types," Amsterdam, The Netherlands, The Netherlands, Tech. Rep., 1999.

[14] A. van Deursen and L. Moonen, "An empirical study into COBOL type inferencing," *Sci. Comput. Program.*, vol. 40, no. 2-3, pp. 189–211, 2001. [Online]. Available: http://dx.doi.org/10.1016/S0167-6423(01)00015-6

[15] R. Komondoor, G. Ramalingam, S. Ch, and J. Field, "Dependent types for program understanding," in *In Proc. Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2005, pp. 157–173.

[16] R. O'Callahan and D. Jackson, "Lackwit: a program understanding tool based on type inference," in *ICSE '97: Proceedings of the 19th international conference on Software engineering*. New York, NY, USA: ACM, 1997, pp. 338–348.

[17] B. Spasojević, M. Lungu, and O. Nierstrasz, "Mining the ecosystem to improve type inference for dynamically typed languages," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! '14. New York, NY, USA: ACM, 2014, pp. 133–142. [Online]. Available: http://scg.unibe.ch/archive/papers/Spas14c.pdf

[18] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, L. Spoon, E. Stenman, and M. Zenger, "An Overview of the Scala Programming Language (2. edition)," École Polytechnique Fédérale de Lausanne, Tech. Rep., 2006.

[19] B. C. Pierce and D. N. Turner, "Local type inference," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 1, pp. 1–44, Jan. 2000. [Online]. Available: http://doi.acm.org/10.1145/345099.345100

[20] D. V. H. Matthew Might, Yannis Smaragdakis, "Resolving and exploiting the k-CFA paradox," in *PLDI*, 2010, pp. 305–315. [Online]. Available: http://matt.might.net/papers/might2010mcfa.pdf

[21] A. H. Borning and D. H. H. Ingalls, "A type declaration and inference system for Smalltalk," in *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '82. New York, NY, USA: ACM, 1982, pp. 133–141. [Online]. Available: http://doi.acm.org/10.1145/582153.582168

[22] N. Suzuki, "Inferring types in Smalltalk," in *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM Press, 1981, pp. 187–199.

[23] R. E. Johnson, "Type-checking Smalltalk," *SIGPLAN Not.*, vol. 21, no. 11, pp. 315–321, Jun. 1986. [Online]. Available: http://doi.acm.org/10.1145/960112.28728

[24] M. Salib, "Faster than C: Static type inference with Starkiller," in *in PyCon Proceedings, Washington DC*. SpringerVerlag, 2004, pp. 2–26.

[25] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks, "Static type inference for Ruby," in *Proceedings of the 2009 ACM Symposium on Applied Computing*, ser. SAC '09. New York, NY, USA: ACM, 2009, pp. 1859–1866. [Online]. Available: http://doi.acm.org/10.1145/1529282.1529700

[26] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, vol. 17, pp. 348–375, 1978.

[27] R. Cartwright and M. Fagan, "Soft typing," in *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1991, pp. 278–292.

[28] F. Pluquet, A. Marot, and R. Wuyts, "Fast type reconstruction for dynamically typed programming languages," in *DLS '09: Proceedings of the 5th symposium on Dynamic languages*. New York, NY, USA: ACM, 2009, pp. 69–78.