Exploiting Type Hints in Method Argument Names to Improve Lightweight Type Inference

Nevena Milojković, Mohammad Ghafari, Oscar Nierstrasz

Exploiting Type Hints in Method Argument Names to Improve Lightweight Type Inference

Nevena Milojković, Mohammad Ghafari, Oscar Nierstrasz
Software Composition Group, University of Bern
Bern, Switzerland
{nevena, ghafari}@inf.unibe.ch
scg.unibe.ch

***Abstract*—The lack of static type information is one of the main obstacles to program comprehension in dynamically-typed languages. While static type inference algorithms try to remedy this problem, they usually suffer from the problem of false positives or false negatives.**

In order to partially compensate for the lack of static type information, a common practice in dynamically-typed languages is to name or annotate method arguments in such a way that they reveal their expected type, *e.g., aString, anInt*, or *string: String*. Recent studies confirmed that these type annotations are indeed frequently used by developers in dynamically-typed languages.

We propose a lightweight heuristic that uses these hints from method argument names to augment the performance of a static type inference algorithm. The evaluation through a proof-of-concept prototype implemented in Pharo Smalltalk shows that the augmented algorithm outperforms the basic algorithm, and correctly infers types for 81% more method arguments.

***Keywords*-type-inference, dynamically-typed languages, heuristic, type hints**

I. Introduction

While dynamically-typed languages offer a more flexible way to write source code [1] and allow developers to be more expressive [2], [3], it is unquestionable that they pose certain difficulties for program comprehension [4]. Even though developing in dynamically-typed languages cuts in half the time needed to write the code [5], static type information has proven to be of crucial importance to developers during software maintenance [6]. This is even of greater significance when we bear in mind that around 70% of the software life cycle is spent on software maintenance [7]. While working in a dynamically-typed environment, developers spend most of the time on reasoning about software and making a mental model of software behaviour [8]. Static type information constitutes the core needed by the developer to understand the control flow of the system [4]. Already pure type information without static type checking has a significant impact on developer's productivity [9].

Type inference has been of research interest for the last several decades [10], [11], [12], [13], [14], [15], [16], [17], [18], [19]. Some approaches rely purely on static information [10], [13], [11], [17], while others exploit information collected during software execution [12], [14], [18]. However, in order to collect run-time information, a fully (and usually instrumented) running program must be available [18].

Simple approaches, intended to be used by the developer for comprehension purposes, usually analyse variable usage only within its scope of availability, thus they are intraprocedural, and not interprocedural [17]. The problem with such algorithms is that they suffer from *false positives*, *i.e.,* classes that understand the interface of the variable, but do not represent its run-time type. Hence, they *over-approximate* the results.

Other approaches try to model software execution, thus they are sensitive to either control flow or data flow or both [13]. In any case, even these more complex static analyses tend to produce faulty results [20]. In the presence of reflection, or dynamic class loading, these algorithms *under-approximate* the set of possible types for a variable, rather than over-approximating them [20]. This causes certain types to be missed, thus introducing the problem of *false negatives*, *i.e.,* the classes that are not inferred as possible types for a variable, yet represent the variable type at run time. A recent study showed that developers in dynamically-typed languages often use dynamic and reflective features [21], [22], [23], [24], as well as in statically-typed languages [25]. For instance, a reflective method invocation is used in more than 60% of the analysed projects, and for almost all of them it is not possible to discover the actual method being invoked only using simple static analyses [24].

In our experience, many developer communities strongly adhere to certain naming conventions. For example, a common idiom in dynamically-typed languages is to name method arguments (*i.e.,* formal parameters to methods) after their expected type [26], [27], [28]. Type hints in method arguments have a positive impact on program comprehension [9]. This is mainly intended to support the developer's reasoning about the variable, but it is also used by some development tools, *e.g.,* code completion [29]. The usage of identifier names has been explored to suggest a new identifier name [30], [31], and study differences and similarities between method parameter and method argument names [32]. A recent study revealed that this naming pattern is not strictly followed, but it may be successfully upgraded with simple heuristics for about a half of the explored method arguments [33], [34], [35].

We believe that these hints can be of crucial importance for type inference in cases where the type of the variable cannot be statically inferred by traditional approaches. We propose a heuristic that combines an existing type inference algorithm

with type hints from method argument names. Even though these hints may be used by any traditional type inference algorithm, we chose the Cartesian Product Algorithm (CPA) [13], which aims to precisely infer types for method arguments. CPA is a traditional type inference algorithm that was used as the base approach for several other algorithms [36], [37]. We employ an extension of the algorithm that is proposed by Spasojević *et al.* [33] in order to obtain the type information from type hints in method argument names.

We have implemented a proof-of-concept prototype in Pharo,[1] which is a dialect of Smalltalk [38], a highly reflective dynamically-typed language that enables fast and easy implementation of analysis tools [39]. We have used this prototype to evaluate our hypothesis. For CPA, which depends on the control flow analysis, this heuristic showed significant improvement in the number of method arguments for which we were able to correctly infer types, as compared with the types recorded at run time. In particular, the combination of CPA with type hints from method argument names, which we call CPA*, is able to increase the size of the correct call graph by 30%, to analyse 52% more method arguments, and to correctly infer types of up to 81% more of the method arguments.

Structure of the Paper. First we motivate our approach in section II. We present the traditional algorithm that we used for the evaluation, and we explain the proposed heuristic in the section III. We explain our prototype implementation in section IV. Next we evaluate the approach in section V. We discuss possible threats to validity in section VI, before we present the related work in section VII and conclude in section VIII.

II. Motivation

While type hints in identifiers are used in dynamically-typed languages, they are mostly intended to support human reasoning about the software at hand [29], [9]. On the other hand, traditional type inference algorithms usually depend only on the analysis of language constructs, rather than on naming conventions. However, in the situations where the use of reflection is involved, these algorithms lack the information needed to infer types [20]. In the continuation of this section, we emphasise the advantage of type hints for type inference through a real code example taken from Smalltalk,[2] however for the sake of generalisability of our discussion we present this example in pseudo-code.

```
1 class PluggableThreePhaseButtonMorph
2    extends ThreePhaseButtonMorph{
3        var pressedImage;
4        var target;
5        var pressedImageSelector;
6 }
```

Listing 1. A subclass definition with three fields.

Consider lines 1-6 in Listing 1 that define a class named `PluggableThreePhaseButtonMorph`, a subclass of the class `ThreePhaseButtonMorph`. This class is a part of the package `Morphic`, a User Interface construction kit [40] used for graphical representations in Pharo Smalltalk. Morphic is based on the idea that each object (a graphical component) is detachable from its parent, and can be manipulated on its own. The class `PluggableThreePhaseButtonMorph` allows the construction of a button that has three different images: one image for when the button is in *on* state, another when it is in *off* state and the third for when the button is just *pressed*.

Let us imagine that the developer wants to understand the control flow of the class, and how to manipulate one such morph. This class has a field named `pressedImage` (line 3 in Listing 1). In order to understand the implementation of a button, the developer wants to statically infer types of the expressions in the class. Since she does not want an over-approximation of the possible types, she decides to use the CPA. It tracks the control flow of the code and the flow of types from one expression to another, and propagates the types through connections between expressions. CPA needs an entry point — a `main` method — and for that purpose one of the factory methods can be used. These methods are defined on the *class side* of a class. Since everything in Smalltalk is considered to be an object, classes are objects, too. Thus the class side of a class defines methods that may be invoked on the class object. Pharo Smalltalk does not force a developer to write and use a `main` method in order to start a program execution. Any method in the project may be used as an entry point. The usual practice in Smalltalk includes using class side methods as `main` methods.

This factory method will first create an instance of the class `PluggableThreePhaseButtonMorph`, thus CPA will infer the type of this construction call to be of that class. The field `pressedImage` is not defined during object creation in the constructor, thus its value will be `null` at the beginning of the analysis.

```
7  class PluggableThreePhaseButtonMorph{
8     method updatePressedImage(){
9       this.setPressedImage(
10         target.perform(pressedImageSelector));
11    }
12 }
```

Listing 2. An example where static analysis cannot determine the type of a method argument.

CPA will continue to analyse the control flow, and, concurrently, to infer the types of the expressions. During its evaluation, CPA will encounter the method named `updatePressedImage()` in the class `PluggableThreePhaseButtonMorph`, presented in Listing 2. This method has one line of code, and it invokes the setter method for the field `pressedImage` (line 9 in Listing 2). The supposed value of the field `pressedImage` is the return value of the method invocation `target.perform(pressedImageSelector)`. Both `target` and `pressedImageSelector` are also fields of the class.

The code `perform(var)` is a reflective way to invoke a method on the target object with the method name

[1] http://www.pharo.org

[2] http://www.smalltalkhub.com/#!/~Pharo/Pharo60/packages/Morphic-Widgets-Basic

supplied as an argument to `perform(var)`. In this case a method with the name equal to the value of the variable `pressedImageSelector` is invoked on the variable `target`. Let us suppose that CPA was able to determine that the type of the variable `pressedImageSelector` is String. String is the expected type of the argument of the method `perform(var)`. However, even if CPA knows the type of `pressedImageSelector`, it does not know its actual value. Thus, the analysis is not able to compute which method will be invoked on the variable `target`. Recent studies revealed that invoking a method in this manner is quite common in Smalltalk code [24], as well as in other dynamically-typed languages [21], [23] and for almost all of the occurrences, it is not easy to determine the actual value of the parameter statically.

In this case two scenarios are possible: either to assign the type `Object` to the return value of the method invocation `target.perform(pressedImageSelector)`, or to leave the set of inferred types for this expression empty. A usual practice in these situations is to *under-approximate* the results, in the case when the concrete value cannot be determined by the analysis [20]. This means that the set of possible types for the return value of the method invocation `target.perform(pressedImageSelector)` will be empty. Hence, in its next step, when the setter method `setPressedImage(var)` (in the Listing 3) is entered, CPA will not know the type of its argument `aForm`. The method implementation consists of one line of code in which the method argument is assigned to the field `pressedImage` (line 15 in Listing 3). CPA works in such a way that it propagates all the inferred types of the right-hand side of an assignment to the variable on the left-hand side of the assignment. Since it did not infer types for the argument named `aForm`, subsequently, it will not be able to infer the types of the field `pressedImage`. Closer investigation of the class definition by the authors reveals that the field `pressedImage` is only assigned through this setter method, thus this loss of information will highly impact further analysis.

```
class PluggableThreePhaseButtonMorph{
    method setPressedImage(var aForm){
                pressedImage := aForm;
        }
}
```

Listing 3. The type hint from the argument name can help to detect the type of the method argument and subsequently the type of the field `pressedImage`.

We propose to exploit type hints in method argument names to highlight which classes are expected to represent the argument type(s) at run time. We argue that these hints may improve analysis precision. In the example in Listing 3 the name of the method argument is `aForm`. A Smalltalk idiom is to embed a type hint in method argument name, *i.e.,* to prefix the name of the expected class with the undefined article. For example, if the expected type is `String`, the corresponding argument name should be `aString`.

The analysis of method argument names embedded in Pharo from which the example is taken, will reveal that the expected type of the argument is represented by the class `Form`. It is an object used for holding images. Thus, with this small improvement, the analysis is able to assign the `Form` class to the set of possible types for the field `pressedImage`, and to continue the analysis with this information.

As a result, the developer will be issued with the information that the field `pressedImage` can have the class `Form` as its type. The actual type of the variable at run time is `ColorForm`, which is a subclass of `Form`. However, this is a common situation in object-oriented languages, due to a heavy use of polymorphism [41].

We argue that it is possible to introduce accurate information about the type of a variable with such a heuristic, and that this will provide more insightful information to developers for program comprehension.

III. Algorithm

A. Cartesian Product Algorithm

The Cartesian Product Algorithm is a type inference algorithm developed by Agesen *et al.* [13]. It is implemented on top of the *Basic Type Inference Algorithm* [42] which statically models the control flow at run time. It has been developed and implemented for Self [43], a prototype-based programming language.

The program under analysis is depicted as a graph whose nodes represent program expressions, and directed edges portray run-time type flow. Each node holds the type information, *i.e.,* the set of classes, representing the possible types of the evaluation of a corresponding expression. For example, the node that represents a constructor call for the class `OrderedCollection` will hold as possible types the set with one element, namely the class `OrderedCollection`. If there is an assignment `x:=1` of value `1` to the variable named `x`, the node for this variable will have a class `Integer` as a possible type. Furthermore, if there is an assignment `y:=x`, the algorithm will propagate all the types from the node representing variable `x` to the node representing variable `y`. Hence, the node for variable `y` will also hold the class `Integer` as a possible type.

The algorithm in the original paper [13] is explained for the Self language. Even though its implementation is language-independent, we will follow Agesen's terminology.

Let us denote by E the set of possible expressions in the project. As defined in Equation 1, it consists, respectively, of literals (the set of literals is denoted by L), variables (V), blocks (B), assignments, method invocations and return statements. A block is a lexical closure [44], [29], present in many programming languages.

$$E = L \cup V \cup B \cup \{v := exp \mid v \in V, exp \in E\} \cup \{x \; msg: \; y | x, y \in E\} \cup \{\hat{} exp \mid exp \in E\} \quad (1)$$

The algorithm has three main steps:

1) create a node in the graph for each expression in the program, *e.g.,* variable named `x`, or an assignment `x:=1`
2) initially seed the types to the nodes for which the type can be determined before the analysis starts, *e.g.,* if there is an assignment `x:=1`, we can seed the node for variable `x` with the type `Integer`.
   The nodes that represent literals are seeded with the class that represents the type of the value held by a literal, *e.g.,* the node for the literal `'string'` is seeded with the type `String`.
3) propagate types along the edges between nodes, *e.g.,* if there is an assignment `y:=x`, we can propagate all the possible types from the node representing variable `x` to the node representing variable `y`, thus `y` will also hold the type `Integer`

The algorithm infers types for expressions based on the constraints it creates during the analysis.

Assignment. During the analysis of an assignment expression `v:=exp` the algorithm infers the type for the expression `exp` on the right-hand side of the assignment, and constructs a directed edge from the node representing the expression `exp` to the node representing variable `v`, indicating that all the types inferred for `exp` should also belong to the set of possible types of variable `v`.

Method invocation. When it encounters a method invocation `x msg: y`, CPA creates edges from the arguments of the method invocation, to the formal parameters of the method that is supposed to be invoked, indicating a possible data flow at run time. If any of the argument expressions has more than one possible type, CPA creates a Cartesian product of the sets of possible types for the arguments, and analyses each of the combinations separately. Thus, whenever the algorithm enters a new method during the analysis, its formal parameters have *uniquely* identified types. After the method has been analysed, CPA caches the information about method argument types (including the type of the object on which method is invoked) and return types. Thus, if at any time in the future the same method needs to be analysed with the same argument types, it can just collect the set of the return types from the cached information, without the need to analyse it again. That is how it preserves speed with accuracy [13].

The set of possibly invoked methods is constructed based on the possible types of the method call receiver. If there is more than one possibly invoked method, edges from the message arguments are constructed to all of the corresponding formal method parameters, to ensure that the entire possible data flow is covered. For each method, the corresponding type of the receiver of method call is seeded as the value of `this`. Correspondingly, the type of the method call is the union of the return types of all methods that can be invoked at that call site.

Return statement. The return type of a method is the union of the return types of all return statements `^exp`[3] within the method. The type of the return statement `^exp` is equal to the type of `exp`, the expression that constitutes the statement.

Block closure. A block is a lexical closure [44], [29], used to postpone the execution of the enclosing expressions. A block can access all the variables in the method in which it is defined, *i.e.,* method temporaries and fields of the enclosing class. It can also define its own arguments and temporaries. As for the method invocation, edges are constructed from the expressions supplied as arguments, to the corresponding block parameters. The return value of the block object is the type of the last expression in the block. Implementation issues concerning lexical closures are discussed in the original paper [13].

Since types are only propagated in the direction indicated by an edge, and never removed from the nodes, an inclusive relation, *e.g.,* `types(x)` $\subseteq$ `types(y)` will always hold for a node `x` with a direct edge to node `y`. Whenever a type is propagated down an edge, the propagation continues onward until it reaches the node that already contained that type, so no further propagation is required. This ensures that the analysis will eventually halt. Analysis continues until a fixed point is reached, and no more propagation is needed.

B. Type hints from method argument names

A common practice when writing dynamically-typed code is to name method arguments to provide a hint of the expected type. This practice differs from one language to another: while in Smalltalk the practice is to prefix the expected class name with an article, *e.g.,* `aString`, in Python the practice is to annotate a method argument in a specific manner[4], *e.g.,* `string: String` to indicate that a method argument named `string` expects an object of type `String`. As a consequence different manners of extracting the type hint from a variable name are needed for distinct languages: while in Smalltalk this involves parsing text, in Python this would require annotation analysis.

For this reason, we leave this part of the algorithm as abstract, supposing that in the specific language there is an implementation of a function returning the inferred type based on the argument, *i.e.,* method argument name. We explain in detail the implementation of this function used for evaluation in section IV.

*C. Upgraded CPA - CPA**

We have implemented CPA with one additional step. Whenever CPA would encounter a method to analyse, just before creating the Cartesian product of the sets of possible types for the arguments, the new algorithm would infer argument type(s) for each of the arguments, based on the type hints from the argument name. Then, for each argument, it would take the union of two sets: the set of inferred types by CPA, and the set of types inferred from the argument name. The algorithm would then continue as usual, to create a Cartesian product of the sets of types, and to analyse each combination separately. We call the new algorithm *CPA**.

[3]Self syntax for `return` statement is `^`.

[4]https://www.python.org/dev/peps/pep-0484/#acceptable-type-hints

Expression	Inferred type
$x = y$ $x == y$ $x \sim= y$ $x > y$ $x >= y$ $x < y$ $x <= y$	`{True, False}`
x msg, where x is a class and msg is any of the selectors from the set {`new, new:, basicNew, basicNew:`}	`x`

TABLE I
HEURISTICS USED TO INFER THE TYPE OF THE EXPRESSION

IV. IMPLEMENTATION

In this section we explain in detail the prototype implementation in Smalltalk.

A. CPA

In order to initially seed the types to the nodes, we have used a couple of heuristics to guess the type of the expression result assigned to the variable, as presented in the Table I. The method for identity comparison (*i.e.,* ==) is implemented as a primitive method. Primitive methods are performed directly by the interpreter rather than by evaluating expressions in the method. Essential primitives cannot be performed in any other way. For example, Smalltalk without primitives can move values from one variable to another, but it cannot add two Integers together. Thus, there is no other way to handle these methods. Without the use of these heuristics, the analysis would lose precision. For example, the inferred return type for method == would be the type of the object on which the method is invoked, rather than an instance of the `Boolean` class. The second row in the Table I refers to the primitive methods used to create new instances of a class.

B. Type Hints in Smalltalk

A recent study on the quality and usage prevalence of type hints from method argument names in Smalltalk [33] revealed that type hints are provided only in around 36% of cases. If the expected type of a Smalltalk method argument is `String`, the corresponding argument should be named `aString`. The authors of the study proposed a couple of heuristics in order to improve the algorithm used for type hints. They managed to successfully guess the type for about half of the method arguments throughout the ecosystem. To guess the type of the argument named `arg`, we have used a slightly improved version of the algorithm proposed in this study. It is important to emphasise that the algorithm includes the following steps in the same order, and that the algorithm would proceed to the following step only if the previous step would fail to provide a type:

1) if there is a class in the Pharo image[5] with name matching `arg` (by ignoring upper and lower case differences), that class represents the type of the argument, *e.g.,* argument named `string` would have the type `String`
2) remove everything in the argument name before the first upper case, and match the rest with a class name, *e.g.,* an argument named `aString` would have the type `String`, but also an argument named `whateverString` would have the type `String`
3) if the argument name is `spec`, its supposed type is `MetacelloAbstractVersionConstructor`. `spec` is commonly used to name specifications of Metacello versions. Metacello is a package management system for Monticello, a distributed version control system used for Smalltalk
4) if the argument name matches the regex "`.*(b|B)lock.*`", its type is `BlockClosure` (this class represents a lexical closure object in Smalltalk)
5) if the argument name matches the regex "`.*(o|O)rderedCollection.*`", its type is `OrderedCollection`
6) if the argument name matches the regex "`.*(a|A)rray.*`", its type is `Array`
7) if the argument name matches the regex "`.*(d|D)ictionary.*`", its type is `Dictionary`
8) if the argument name matches the regex "`.*(s|S)et.*`", its type is `Set`
9) if the argument name matches the regex "`.*(b|B)ag.*`", its type is `Bag`
10) if the argument name matches the regex "`.*(c|C)ollection.*`", its type is `Collection`
11) if the argument name matches the regex "`.*(s|S)tring.*`", its type is `String`
12) if the argument name matches the regex "`.*(s|S)ymbol.*`", its type is `Symbol`. Symbols in Smalltalk represent Strings that are created uniquely.

The algorithm in the study performed by Spasojević *et al.* did not include steps *5-9*. Thus, for any method argument whose name matches regular expression "`.*(c|C)ollection.*`" the inferred type would be `Collection`. However, since the `Collection` class is an abstract superclass of the classes `OrderedCollection`, `Array`, `Dictionary`, `Set` and `Bag`, and, based on our analysis of the current Pharo image, these subclasses are the most commonly used subclasses of the `Collection` class, we decided to treat them separately.

It is important for the algorithm to follow the steps in the indicated order, for the sake of the argument names like `aBlockAnsweringAString`, which is clearly a `BlockClosure` and not a `String`, or `aCollectionOfString` which is a `Collection` and not a `String` [33].

If a method argument can expect an object of the type

[5]The term "Pharo image" is used to denote snapshot of the running Pharo system.

Project name	# of methods	# of methods with arguments	# of arguments
Roassal2	1371	483	745
Glamour	189	188	229
Morphic	677	675	935
TOTAL	2237	1346	1909

TABLE II
RUN-TIME INFORMATION

`BlockClosure` or `Symbol`, the convention is to name the argument *e.g.,* `aBlockOrSymbol`. The usual approach is to use the conjunction `Or` starting with a capital letter and followed by a capital letter, due to the Camel Case notation [45]. While this is a convention in Smalltalk, a different approach might be taken in other languages [46]. In order not to lose type hints for these arguments, we would first split the argument name based on the appearances of conjunction `Or` followed by an upper case letter, and then apply steps 1-12 from the algorithm on each of the substrings. The type(s) of the argument would be represented by the union of type(s) of each of the substrings.

The proposed algorithm heavily depends on correctly named method arguments. We match regular expressions that contain the expected class name with the method argument name. Hence, if the method argument name contains an error and, for example, is named `aStrong` instead of `aString`, the algorithm will miss its type.

V. EVALUATION

For the evaluation we have used three open-source Pharo projects for which we were able to collect run-time information that closely depicts their real usage: Glamour[6] [47], Roassal2[7] [48], and Morphic [40]. Glamour is a framework for specifying the navigation flow of browsers. Roassal2 is an agile visualisation engine that graphically renders objects and Morphic is a User Interface construction kit.

Each project provides a set of *example methods* that reflect their real usage: Glamour has 83 of these methods, Morphic 29, and Roassal2 952. These methods are created by project developers to demonstrate the potential usage of the corresponding project. They serve as a `main` method and when executed provide an example of how the project may be employed. We have executed these methods, and the recorded run-time data serves as ground truth to which results provided by static type inference are compared.

As can be seen in Table II, the execution of these examples covered in total 2237 methods from all three projects, out of which 1346 methods have at least one argument. It traversed in total 1909 method arguments. We have recorded run-time information only for the methods in the packages, *i.e.,* we have excluded the library methods.

We have used the same example methods as `main` methods, *i.e.,* the entry points to start the Cartesian product analysis. The overall results are represented in Figure 1. In total, CPA* analysed 31% more methods, which increased the number of analysed arguments by 52%. Consequently, the number of method arguments that contain type hints was also increased by 63%. However, 22 out of 212 method arguments contained false type information, that is the type inferred from the argument name did not correspond to its run-time type. CPA* is nevertheless able to correctly infer types for 81% more of the method arguments.

More detailed information regarding CPA and CPA* separately are presented in Table III and Table IV. When we executed the CPA analysis in its basic form, in our implementation it managed to cover 316 out of 1371 methods in the Roassal2 package, or one quarter of the actual call graph. As for the Glamour package, analysis covered 62 methods, *i.e.,* around 33% of the executed methods. We were surprised by the low number of statically analysed methods in the Morphic project. CPA managed to reach only 3% of the executed methods. The Morphic project takes advantage of the reflection usage. It invokes the `Object»#perform:`[8] method in 29 of its methods. The method `Object»#perform:` provides a reflective way to invoke a method on the target object with the method name supplied as an argument to `Object»#perform:`. After having a closer look at the analysis, we discovered that 12 of these 29 methods were traversed during static analysis and that they were traversed in total a bit more than 3000 times. This puts them in the hotspot of the analysis. Two of these twelve methods do not have any method argument, thus there is no space for improvement. As for ten other methods, almost no type hints may be found in their argument names. The same applies for the methods that are being traversed right after the `Object»#perform:` methods, hence room for improvement in this case is limited.

These 399 methods account for 253 method arguments. For 45% of these arguments, *i.e.,* for 114 out of 253, all types seen at run time were also inferred by CPA (this is presented in the column named "# of covered arguments" in Table III). Hence, CPA did not underestimate types for a bit less than half of the arguments. The proportion of arguments with type hints in their name is quite different in all three packages: from 40% in Roassal2 to 83% in Morphic.

When we inferred types with the CPA* algorithm, we were able to cover 123 more methods, *i.e.,* 31% (Table IV). This indicates that 130 arguments in CPA analysis with type hints in their name managed to augment the size of analysed call graph for around one method per argument with type hint. We deem this important, since it increases the size and precision of the traversed call graph.

Accordingly, the number of analysed method arguments increased by 52%. The number of arguments for which all run-time types were also inferred by static analysis increased to 53.5%. The augmented algorithm therefore outperforms the basic algorithm, and correctly infers types for 81% more method arguments.

[6]http://www.smalltalkhub.com/#!/~Moose/Glamour

[7]http://smalltalkhub.com/#!/~ObjectProfile/Roassal2

[8]The `perform:` method of the `Object` class

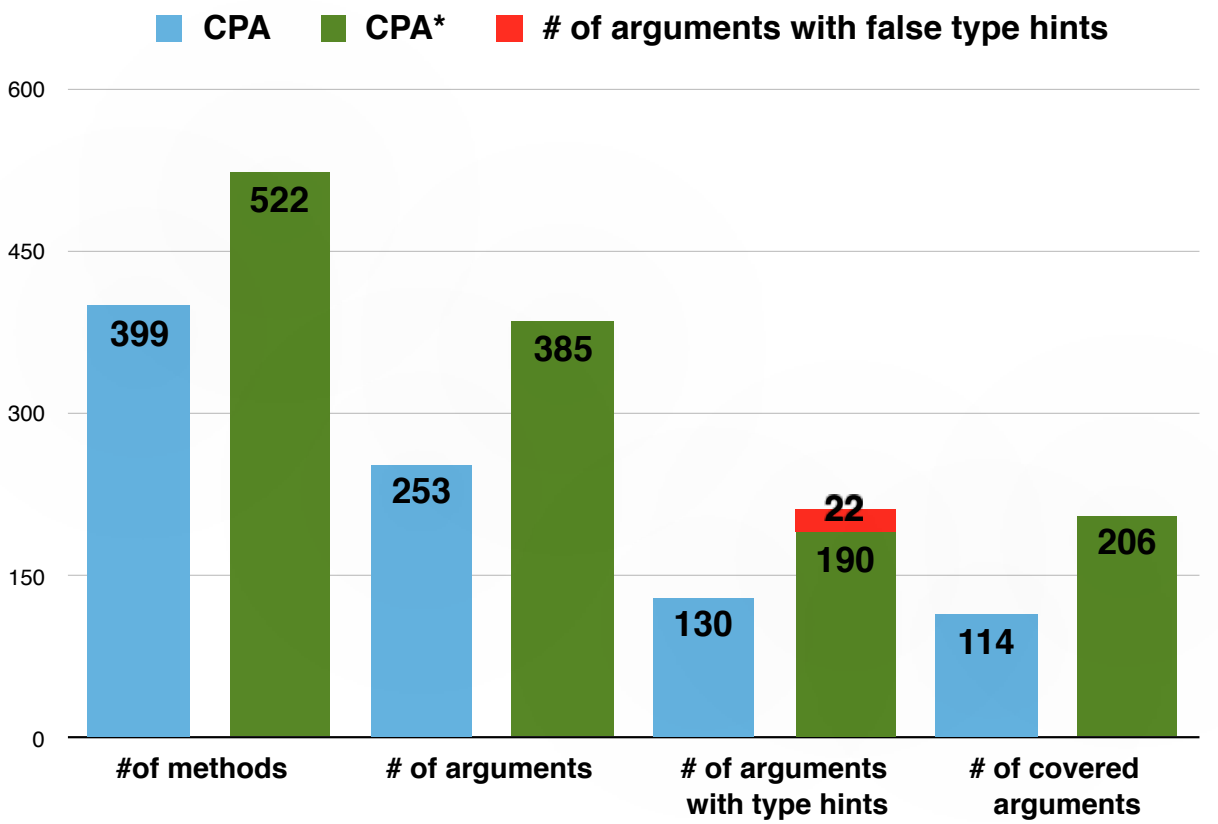

Fig. 1. CPA and CPA* results

Project name	# of methods	# of methods with arguments	# of arguments	# of arguments with type hints	# of covered arguments
Roassal2	316	114	155	63	63
Glamour	62	62	75	48	41
Morphic	21	21	23	19	10
TOTAL	399	197	253	130	114

TABLE III
CPA BASIC

Project name	# of methods	# of methods with arguments	# of arguments	# of arguments with type hints	# of arguments with false type hints	# of covered arguments
Roassal2	383	146	224	95	21	133
Glamour	66	66	80	50	0	46
Morphic	73	73	81	67	1	27
TOTAL	522	285	385	212	22	206

TABLE IV
CPA WITH TYPE HINTS - CPA*

The column named "# of arguments with false type hints" in Table IV provides information about misleading type hints. Most of them are due to the name `aCollection` for which only the class `Collection` can be inferred as type, and since `Collection` is an abstract class, it is not used as object type at run time. Also the argument name `aShape` is misleading, as the `Shape` class will be inferred as the possible type, although the run-time type is actually `RTShape`, which is not related to the `Shape` class even though it has a similar name.

There are a couple of situations where CPA* would infer as possible types for a method argument both `BlockClosure` and `Symbol`, while at run time only one of them is recorded as the actual type. This would happen if a method argument is named *e.g.,* `aBlockOrASymbol`. Such variables often

Project name	Time - CPA	Time - CPA and type hints
Roassal2	11.5	12.1
Glamour	0.7	0.9
Morphic	1	1.6

TABLE V
TIME NEEDED FOR THE ANALYSIS IN SECONDS

reflect a form of *duck typing* [49]. In this specific case, for example, `Symbol` is duck-typed in Smalltalk to behave like a block in certain idiomatic scenarios. We therefore do not consider CPA* to be wrong just because only one of the two types is observed in practice. Also, if CPA* would infer as possible types for a method argument more classes than recorded at run-time, but the argument name clearly reveals that any of the inferred classes is expected at run time, *e.g.,* `aBlockOrANumber`, we do not count it as false type hint. We choose to believe in the developer's suggestions, even though not all of the hinted classes are recorded during the execution run.

This data shows that the combination of CPA with type hints from method argument names significantly increased the size of the analysed call graph, which forms a part of call graph recorded at run time, thus does not belong to false positives. This indicates that type hints from method argument names can increase the precision of CPA in dynamically-typed languages.

In order to evaluate whether or not this combination is still usable for program comprehension purpose, beside the precision, we have measured the time needed for both types of the analysis. On all three projects, we have found that the introduced overhead is quite divergent: from 5% to 60% (Table V). The most overhead was introduced in the Morphic project, which is understandable, since the number of analysed methods more than tripled. This also indicates that Morphic contains method arguments with the largest number of type hints among the three analysed packages.

A. Argument names without inferred type hint

In all three projects, we have investigated the set of argument names for which the algorithm was not able to provide a hint. In this subsection, we elaborate on our findings.

Roassal2. The most used names are, in order, `elements`, `objects`, `anElement`, `anObject`, `element`. Even though the analysis was not able to infer types from these names, a human can deduce that the first two arguments expect some kind of `Collection` object. Even though we think that this can be introduced as a rule, more research is needed to verify our assumption. As for the third name, there is no class named `Element`, but there is a class in the Roassal2 project named `RTElement`, so a developer may guess the expected type of the argument. The same applies for the argument named `element`. The name `anObject` is used as a method argument name in all three projects: in Roassal2 it is mostly used to indicate a member of a collection, or as an object on top of which a graphical representation is built; while in Glamour and Morphic it is mostly used as a name of a setter-method argument.

Glamour. Beside `anObject` the most common argument names for which the algorithm was not able to provide a type hint are `aPort`, `aPortReference`, `aPane` and `aPresentation`. These names correspond to types from the Glamour project, respectively, `GLMPort`, `GLMPortReference`, `GLMPane` and `GLMPresentation` (GLM stands for Glamour), which indicates that this kind of heuristic would greatly benefit from the input related to the project under analysis. Based on the examples from the Roassal2 and Glamour packages, the simple heuristic of removing the project-related prefix from class names and then inferring a type from the argument name would definitely improve the results. However, the improvement heavily depends on the practice of developers involved in the development of certain project. In these particular projects, we suspected that the improvement would be significant, since about 30% of method arguments without type hints in Roassal2 and 50% of those in Glamour would provide a type hint by using this heuristic. Further investigation revealed that these classes are mostly abstract. If we have a bounded set of possible concrete classes in this case, inferring the actual type of such arguments would be feasible by applying an approach similar to the one we followed to detect the type for collections, as was explained in subsection IV-B.

Morphic. An argument named `anImage`, which would suggest that an object of class `Image` is expected, actually expects an object of class `Form` which represents an array of pixels, used for holding images. No class `Image` exists in the version of Pharo used for the experiment. A method argument named `aFont` indicates that some kind of `font` is expected at run time. While there is no class `Font`, there is an abstract class named `AbstractFont`, which defines the interface for `fonts`.

Findings in the Glamour project cause us to suppose that giving precedence to classes from the same package when analysing a method would improve the results. For example, a method argument `anAnnouncer` clearly indicates that it expects some kind of the announcer at run time. While the used algorithm for inferring the type from method argument name would conclude that its type is `Announcer`, in Glamour package its expected type is actually `GLMAnnouncer`, a subclass of the `Announcer` class.

VI. DISCUSSION AND THREATS TO VALIDITY

We have chosen the *Roassal2, Glamour* and *Morphic* projects to evaluate our heuristic since we were able to run these projects in a way that closely resembles their real usage. Nevertheless, it is an open question whether we have collected all possible run-time types for variables.

Another threat to validity comes from the quality of type hints in method argument names. While we have evaluated our heuristic in Smalltalk, exploring its performance in other dynamically-typed languages remains future work. Recent

studies revealed that type annotations are commonly used in Python [34] and Dart [35]. Hence, we believe that work similar to ours may be performed at least in these two languages.

Moreover, we have used the results provided by a study on a large set of Smalltalk projects [33], but it is an open question whether an arbitrary project will provide the same quality of type hints. As is presented in the section V, some argument names have misleading type hints, while others provide a type hint for an abstract class, but it is obvious that a subclass will actually represent an argument type. Due to this problem, we have enriched the algorithm used to obtain a type from an argument name by steps 5-9 (subsection IV-B). The same kind of work is possible *e.g.,* for the argument name `anAbstractFont` (this name indicates an abstract class, while the usual run-time type is one represented by one of the subclasses), but we did not have any findings on the larger set of projects to support this.

The final threat comes from our choice of the basic algorithm. We chose CPA since its precision heavily depends on the correctly inferred types for method arguments and we think that it would greatly benefit from the proposed heuristic. However, we believe that this heuristic may be also combined with other simple type inference algorithms, for example RoelTyper [17]. The heuristic is simple and fast enough not to endanger the speed of the underlying algorithm, yet we assume it would improve its precision.

VII. Related work

To the best of our knowledge, this is the first study that tries to combine an existing type inference algorithm with type hints from method argument names.

The first type inference algorithm, "Algorithm W", was developed by Milner [10]. It was developed for ML programming language, and focused on parametric types rather than subtyping. Types are considered to be hierarchies of functional types over a set of basic types.

Another approach that treats types as sets of expressions, rather than as sets of values was developed for the FL functional language [11]. We focus our attention on algorithms that employ nominal types, since structural types burden program comprehension [50].

One of the most precise type inference algorithms, *Chuck* was developed by Alexander Spoon *et al.* [16], [51]. It is a demand-driven algorithm that uses *subgoal pruning*. That means that if the type inference for an expression, *i.e.,* the goal, would depend on many other complex expressions, *i.e.,* type inference of subgoals, the algorithm may decide to *prune* some of the subgoals in order to simplify the calculations.

The Cartesian Product Algorithm [13], which we used as the underlying algorithm in our implementation, was originally developed for *Self*, a prototype-based language [43]. It has been used as the basic algorithm for a couple of other type inference techniques [15], [36], [37]. *Starkiller* represents a type inferencer and compiler for Python, based on CPA [15]. Another extension of CPA is DCPA [36], which focuses on *data polymorphism*, *i.e.,* when a variable may have assigned values of different types. It was used to check the correctness of downcasts in Java programs. *Ecstatic* [37] is an implementation of type inference for the Ruby programming language based on CPA, and adapted for various Ruby programming idioms, for example block usage, as there are seven distinct block syntaxes.

One of the simplest type inference algorithms, *RoelTyper*, which is mainly intended for the purpose of program comprehension, was developed by Roel Wuyts and later improved by Frédéric Pluquet [17]. The algorithm is intraprocedural, and it infers types for a variable based on its interface and possible assignments to it. Even though it is very simple, it correctly infers types for just under 60% of variables.

One improvement of RoelTyper was presented by Spasojević *et al.* [52]. It collects information from the language ecosystem, hence the name *Ecosystem-Aware Type Inference (EATI)*. Based on the collected information, EATI sorts possible types for a variable and increases the likelihood of putting the correct type at the top of the list of possible types.

RoelTyper was improved by heuristics that derive data from static and dynamic analysis of class usage frequency [53], [54]. These heuristics more than doubled the number of correctly inferred types when compared to the underlying approach.

Another related field of work is optional typing. It grants a developer the opportunity to annotate types, but does not enforce static type checking, hence does not prevent run-time type errors [55]. Optional typing is present in many language communities: Smalltalk [56], [57], [58], [59], Groovy [60], Dart [35], JavaScript [61], Python [62], Ruby [63] and Lua [64].

These type annotations usually serve as a kind of documentation and they are often exercised by developers [35], [65]. Developers with less background in dynamically-typed languages tend to use type annotations more often, mostly in method definitions. Furthermore, types are less used in frequently changed code. As seen, type annotations may serve as input for improvement of type inference algorithms, as presented in this paper.

VIII. Conclusion

Static type information proves to be useful when performing program comprehension tasks in dynamically-typed languages. Current type inference algorithms usually suffer from the problems of false positives or false negatives. However, they mainly analyse language constructs, not naming conventions.

On the other hand, to partially remedy the lack of static type information in dynamically-typed languages, developers tend to provide type hints in the method argument names. These hints are mostly intended for human comprehension, but are also exploited by the IDE, for example, for code completion.

We have performed a study to assess the possible impact of these hints on the results of a type inference algorithm called CPA. The obtained results are promising; the augmented algorithm outperforms the basic one significantly.

ACKNOWLEDGMENTS

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Analysis" (SNSF project No. 200020-162352, Jan 1, 2016 - Dec. 30, 2018). We also gratefully acknowledge the financial support of the Swiss Group for Object-Oriented Systems and Environments (CHOOSE) and the European Smalltalk User Group (ESUG). We thank Michael Jungo for the source code that we used as a starting point for our CPA implementation.

REFERENCES

[1] O. L. Madsen, B. Magnusson, and B. Moller-Pedersen, "Strong typing of object-oriented languages revisited," in *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, vol. 25, Oct. 1990, pp. 140–150.

[2] J. K. Ousterhout, "Scripting: Higher level programming for the 21st century," *IEEE Computer*, vol. 31, no. 3, pp. 23–30, Mar. 1998. [Online]. Available: http://www.cs.indiana.edu/classes/c102/read/Ousterhout.pdf

[3] 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

[4] J. Kubelka, A. Bergel, and R. Robbes, "Asking and answering questions during a programming change task in the Pharo language," in *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU '14. New York, NY, USA: ACM, 2014, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/2688204.2688212

[5] 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

[6] S. Kleinschmager, S. Hanenberg, R. Robbes, E. Tanter, and A. Stefik, "Do static type systems improve the maintainability of software systems? An empirical study," in *2012 IEEE 20th International Conference on Program Comprehension (ICPC)*, Jun. 2012, pp. 153 –162.

[7] B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Computer*, vol. 34, no. 1, pp. 135–137, Jan. 2001. [Online]. Available: http://dx.doi.org/10.1109/2.962984

[8] R. Minelli, A. M. and, and M. Lanza, "I know what you did last summer: An investigation of how developers spend their time," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ser. ICPC '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 25–35. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820282.2820289

[9] S. Spiza and S. Hanenberg, "Type names without static type checking already improve the usability of APIs (as long as the type names are correct): An empirical study," in *Proceedings of the 13th International Conference on Modularity*, ser. MODULARITY '14. New York, NY, USA: ACM, 2014, pp. 99–108. [Online]. Available: http://doi.acm.org/10.1145/2577080.2577098

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

[11] A. Aiken and B. Murphy, "Static type inference in a dynamically typed language," in *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '91. New York, NY, USA: ACM, 1991, pp. 279–290. [Online]. Available: http://doi.acm.org/10.1145/99583.99621

[12] U. Hölzle and D. Ungar, "Optimizing dynamically-dispatched calls with run-time type feedback," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI '94. New York, NY, USA: ACM, 1994, pp. 326–336.

[13] O. Agesen, "The Cartesian product algorithm," in *Proceedings ECOOP '95*, ser. LNCS, W. Olthoff, Ed., vol. 952. Aarhus, Denmark: Springer-Verlag, Aug. 1995, pp. 2–26.

[14] P. Rapicault, M. Blay-Fornarino, S. Ducasse, and A.-M. Dery, "Dynamic type inference to support object-oriented reengineering in Smalltalk," in *Proceedings of the ECOOP '98 International Workshop Experiences in Object-Oriented Reengineering, abstract in Object-Oriented Technology (ECOOP '98 Workshop Reader forthcoming LNCS)*, 1998, pp. 76–77. [Online]. Available: http://scg.unibe.ch/archive/famoos/Rapi98a/type.pdf

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

[16] S. A. Spoon and O. Shivers, "Demand-driven type inference with subgoal pruning: Trading precision for scalability," in *Proceedings of ECOOP'04*, 2004, pp. 51–74.

[17] 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.

[18] D. An, A. Chaudhuri, J. Foster, and M. Hicks, "Dynamic inference of static types for Ruby," in *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)*. ACM, 2011, pp. 459–472.

[19] E. Allende, O. Callaú, J. Fabry, É. Tanter, and M. Denker, "Gradual typing for Smalltalk," *Science of Computer Programming*, Aug. 2013. [Online]. Available: http://hal.inria.fr/hal-00862815

[20] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: A manifesto," *Commun. ACM*, vol. 58, no. 2, pp. 44–46, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2644805

[21] A. Holkner and J. Harland, "Evaluating the dynamic behaviour of Python applications," in *Proceedings of the Thirty-Second Australasian Conference on Computer Science-Volume 91*. Australian Computer Society, Inc., 2009, pp. 19–28.

[22] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," *SIGPLAN Not.*, vol. 45, no. 6, pp. 1–12, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1809028.1806598

[23] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do: A large-scale study of the use of eval in JavaScript applications," in *Proceedings of the 25th European Conference on Object-oriented Programming*, ser. ECOOP'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 52–78. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032497.2032503

[24] O. Callaú, R. Robbes, É. Tanter, and D. Röthlisberger, "How (and why) developers use the dynamic features of programming languages: the case of Smalltalk," *Empirical Software Engineering*, 2013. [Online]. Available: http://pleiad.dcc.uchile.cl/papers/2013/callauAl-emse2013.pdf

[25] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *2011 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 241–250.

[26] 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

[27] M. Zandstra, *PHP Objects, Patterns, and Practice*, 4th ed. Berkely, CA, USA: Apress, 2013.

[28] M. Bolin, *Closure: The Definitive Guide: Google Tools to Add Power to Your JavaScript*. O'Reilly Media, 2010. [Online]. Available: https://books.google.ch/books?id=p7uyWPcVGZsC

[29] 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

[30] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 281–293. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635883

[31] ——, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 38–49. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786849

[32] H. Liu, Q. Liu, C.-A. Staicu, M. Pradel, and Y. Luo, "Nomen est omen: Exploring and exploiting similarities between argument and parameter names," in *International Conference on Software Engineering (ICSE)*, 2016.

[33] B. Spasojević, M. Lungu, and O. Nierstrasz, "A case study on type hints in method argument names in Pharo Smalltalk projects," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, Mar. 2016, pp. 283–292. [Online]. Available: http://scg.unibe.ch/archive/papers/Spas16a.pdf

[34] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python probabilistic type inference with natural language support," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 607–618. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950343

[35] M. Faldborg, T. L. Nielsen, and B. Thomsen, "Type systems and programmers: A look at optional typing in Dart," Master's thesis, Aalborg University, 2015.

[36] T. Wang and S. F. Smith, "Precise constraint-based type inference for Java," in *Proceedings ECOOP '01*, ser. LNCS, G. Goos, J. Hartmanis, and J. van Leeuwen, Eds., vol. 2072. Budapest, Hungary: Springer-Verlag, Jun. 2001, pp. 99–118.

[37] M. Madsen, P. Sørensen, and K. Kristensen, "Ecstatic–type inference for Ruby using the Cartesian product algorithm," Master's thesis, Aalborg University, 2007.

[38] 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

[39] B. Foote and R. E. Johnson, "Reflective facilities in Smalltalk-80," in *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, vol. 24, Oct. 1989, pp. 327–336.

[40] H. Fernandes and S. Stinckwich, "Morphic, les interfaces utilisateurs selon Squeak," Jan. 2007.

[41] N. Milojković, A. Caracciolo, M. Lungu, O. Nierstrasz, D. Röthlisberger, and R. Robbes, "Polymorphism in the spotlight: Studying its prevalence in Java and Smalltalk," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. IEEE Press, 2015, pp. 186–195, published. [Online]. Available: http://scg.unibe.ch/archive/papers/Milo15a.pdf

[42] J. Palsberg and M. I. Schwartzbach, "Object-oriented type inference," in *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, vol. 26, Nov. 1991, pp. 146–161. [Online]. Available: http://www.cs.purdue.edu/homes/palsberg/publications.html

[43] D. Ungar and R. B. Smith, "Self: The power of simplicity," in *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, vol. 22, Dec. 1987, pp. 227–242.

[44] M. Pilgrim, *Dive Into Python*. APress, 2004.

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

[46] E. Hill, D. Binkley, D. Lawrie, L. Pollock, and K. Vijay-Shanker, "An empirical study of identifier splitting techniques," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1754–1780, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10664-013-9261-0

[47] P. Bunge, "Scripting browsers with Glamour," Master's Thesis, University of Bern, Apr. 2009. [Online]. Available: http://scg.unibe.ch/archive/masters/Bung09a.pdf

[48] 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.

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

[50] "The Strongtalk type system for Smalltalk," http://bracha.org/nwst.html. [Online]. Available: http://bracha.org/nwst.html

[51] S. A. Spoon and O. Shivers, "Dynamic data polyvariance using source-tagged classes," in *Proceedings of the Dynamic Languages Symposium'05*, R. Wuyts, Ed. ACM Digital Library, 2005, pp. 35–48.

[52] 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

[53] N. Milojković, C. Béra, M. Ghafari, and O. Nierstrasz, "Inferring types by mining class usage frequency from inline caches," in *Proceedings of International Workshop on Smalltalk Technologies (IWST 2016)*, 2016, pp. 6:1–6:11. [Online]. Available: http://scg.unibe.ch/archive/papers/Milo16a.pdf

[54] N. Milojković and O. Nierstrasz, "Exploring cheap type inference heuristics in dynamically typed languages," in *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016*. New York, NY, USA: ACM, 2016, pp. 43–56. [Online]. Available: http://scg.unibe.ch/archive/papers/Milo16b.pdf

[55] G. Bracha, "Pluggable type systems," in *In OOPSLA'04 Workshop on Revival of Dynamic Languages*, Oct. 2004. [Online]. Available: http://prog.vub.ac.be/~wdmeuter/RDL04/papers/Bracha.pdf

[56] J. O. Graver and R. E. Johnson, "A type system for Smalltalk," in *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '90. New York, NY, USA: ACM, 1990, pp. 136–150. [Online]. Available: http://doi.acm.org/10.1145/96709.96722

[57] G. Bracha and D. Griswold, "Strongtalk: Typechecking Smalltalk in a production environment," in *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, vol. 28, Oct. 1993, pp. 215–230. [Online]. Available: http://bracha.org/oopsla93.ps

[58] R. Smit, "Pegon," https://sourceforge.net/projects/pegon/.

[59] N. Haldimann, M. Denker, and O. Nierstrasz, "Practical, pluggable types," in *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*. ACM Digital Library, 2007, pp. 183–204. [Online]. Available: http://scg.unibe.ch/archive/papers/Hald07b-Typeplug.pdf

[60] V. Subramaniam, *Programming Groovy 2: Dynamic Productivity for the Java Developer*, ser. Pragmatic Bookshelf. Pragmatic Bookshelf, 2013, no. v. 2. [Online]. Available: https://books.google.ch/books?id=jvbTmAEACAAJ

[61] G. Bierman, M. Abadi, and M. Torgersen, "Understanding TypeScript," in *ECOOP 2014 – Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28 – August 1, 2014. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 257–281. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-44202-9_11

[62] M. M. Vitousek, A. M. Kent, J. G. Siek, and J. Baker, "Design and evaluation of gradual typing for Python," *SIGPLAN Not.*, vol. 50, no. 2, pp. 45–56, Oct. 2014. [Online]. Available: http://doi.acm.org/10.1145/2775052.2661101

[63] M. Furr, "Combining static and dynamic typing in Ruby," Ph.D. dissertation, University of Maryland, 2009. [Online]. Available: https://www.cs.umd.edu/~jfoster/papers/thesis-furr.pdf

[64] A. M. Maidl, F. Mascarenhas, and R. Ierusalimschy, "Typed Lua: An optional type system for Lua," in *Proceedings of the Workshop on Dynamic Languages and Applications*, ser. Dyla'14. New York, NY, USA: ACM, 2014, pp. 3:1–3:10. [Online]. Available: http://doi.acm.org/10.1145/2617548.2617553

[65] C. Souza and E. Figueiredo, "How do programmers use optional typing? An empirical study," in *Proceedings of the 13th International Conference on Modularity*, ser. MODULARITY '14. New York, NY, USA: ACM, 2014, pp. 109–120. [Online]. Available: http://doi.acm.org/10.1145/2577080.2582208