Exploring Cheap Type Inference Heuristics in Dynamically Typed Languages

Nevena Milojković
SCG, University of Bern
Switzerland
nevena@inf.unibe.ch

Oscar Nierstrasz
SCG, University of Bern
Switzerland
scg.unibe.ch

Abstract

Although dynamically typed languages allow developers to be more productive in writing source code, their lack of information about types of variables is one of the main obstacles during program comprehension. Static type information helps developers to decrease software maintenance time. Inference of types of variables requires complex algorithms to avoid false positives or negatives. Their main aim is to shorten the list of possible types for a variable.

We have developed a couple of cheap heuristics that use easily accessible information about the presence of each class in the available source code to promote the correct type towards the top of the list of possible types for a variable.

Our evaluation of a proof-of-concept prototype, implemented in Pharo Smalltalk, shows that both for domain-specific types and standard libraries these heuristics tend to work well. The evaluated heuristics prove to be reasonably precise, promoting the correct types of a variable towards the top of the list in 50.67% up to 89.09% of cases on average, depending on the applied heuristic. The heuristic that has proved to be the best was compared with one existing type inference algorithm and the best heuristic yields significantly better results with less effort.

Categories and Subject Descriptors D.2.3 [*Coding Tools and Techniques*]: Object-oriented programming; D.3.3 [*Language Constructs and Features*]: Data types and structures

Keywords type inference, heuristics, dynamic languages

1. Introduction

While the presence of a static type system decreases development time [25], the expressiveness of dynamically typed languages improves programmer productivity [22, 30]. Conversely, code written in dynamically typed languages is more error-prone than statically typed code [18].

The benefit of static type information has also been measured with regard to software maintainability [2, 26]. Static type systems are (unsurprisingly) useful for fixing type errors [23]. The lack of static type information in dynamically typed languages can impede the process of program understanding [10, 24]. This is important since developers spend more than two thirds of their time on maintaining software [6], mostly on program comprehension [29].

There have been many attempts to statically infer types of variables in dynamically typed languages [1, 4, 27, 31, 32, 36, 41]. Most of them rely on the set of messages sent to the variable, *i.e.*, tracking down all messages sent to the variable of interest and then determining which classes implement those methods, or inherit them from their superclass. Some of the algorithms are flow-insensitive [31], while others demand more complex calculations, because they take control and data flow of the program into account [1]. The problem is to determine which classes understand the interface of a variable. These classes represent the potential types of the variable. The problem with such approaches is that they suffer from "false positives", *i.e.*, classes that understand the interface of the variable, but do not represent the run-time type of the variable. There exist more complicated algorithms that analyse the data and control flow of the program, but they still suffer from false positives.

The problem lies in the length of the list of possible types. The aim of the existing type inference algorithms is to narrow down this list to the types possibly bound to the variable at run-time. In order to shorten the list and make it more precise, more time- and space-consuming algorithms must be applied. These approaches are not intended to be used by developers for understanding purpose, since they demand a great deal of time to be precise [31]. Ultimately, both simple and complex approaches suffer from the same problem. Presenting the list of possible types for a variable in no particular order, where most of them cannot actually

represent the variable type during run time, is not helpful to a developer.

During the evaluation process we have found that more than half of the analyzed variables have a statically inferred type which may belong to more than 3 different class hierarchies, including 152 classes. All of these classes are more or less likely to represent the actual type of variable during run time. We believe that this likelihood is related to the degree of class usage throughout the source code. Classes are used to instantiate a new object, or to invoke class methods that perform utility functions. Our approach is to order the list of possible types of a variable according to how frequently those classes occur in the source code. We hypothesise that class usage frequency serves as a reliable proxy for the likelihood that a variable belongs to that class at run time. The information about class presence in the source code is easily retrieved, thus preserving the implemented approach as simple and swift, producing the results on average in 0.13 seconds. The developed approach is intended mainly as a tool for program comprehension, to be used by developers during software maintenance.

We have implemented the proof-of-concept prototype and used it to evaluate our heuristics. The heuristic that showed the best results was compared to an existing type inference technique, and showed better results with fewer requirements. The improvement is achieved in 58.6% of cases.

Structure of the Paper. We start by giving an overview of the problem in section 2. We discuss the related work in the field in section 3. Next we define the used terminology and implemented heuristics in section 4; section 5 shows results of the evaluation of the prototype. We then describe potential threats to the validity in section 6 before concluding in section 7.

2. Overview

To better understand the contribution of the paper, let us consider the example in Listing 1. The example[1] is written in Pharo Smalltalk.[2]

```
GLMLoggedObject subclass: #GLMPane
        instanceVariableNames: '... presentations ...'
        classVariableNames: ''
        category: 'Glamour-Core'

GLMPane>>addPresentationSilently: each
        ^ presentations
                add: (each pane: self; yourself)
```

Listing 1. The run-time type of the argument "**each**" cannot be statically detected by the traditional approach

[1] This code snippet is actual code from the Glamour-Core system: `http://www.smalltalkhub.com/#!/~Moose/Glamour/packages/Glamour-Core`

[2] `http://www.pharo.org`. Pharo is a Smalltalk IDE, including a large library that contains the core of the Smalltalk system itself.

Lines 1-4 define a new class GLMPane which has an instance variable presentations, while the set of lines 6-8 define a method named addPresentationSilently: with argument each. This method sets the pane of the argument each to be the GLMPane object itself (pane: is a setter method), then adds the value (yourself) of each to the variable presentations. Finally the result of the computation is returned (^ is the return operator in Smalltalk).

Let us suppose that the developer wants to know the possible type of the argument each in Listing 1, as she is interested to know the types of the individual elements added to the instance variable presentations. Since Smalltalk is a dynamically typed language, the developer cannot determine statically the potential type of the argument each.

The simple (standard) approach to infer variable types for this variable would be to traverse the list of messages sent to it[3] and find all the classes in the image[4] that understand the interface of the variable, *i.e.*, in this case "pane:" and "yourself" selectors. Those classes would be presented in no particular order.

Using the traditional approach, the developer will be offered the list of 75 possible classes[5] grouped by their hierarchies. Thus, she will obtain a list of 75 possibilities, with no particular knowledge of how likely it is that any of the classes is the correct one. Evidently, the existing source of information is not helpful to the developer. Even if the developer would restrict the search of the types to the package in which the class GLMPane occurs, it would still leave her with 50 classes to examine.

To narrow down the list, we note that each will be made an element of presentations, which is presumably a collection. By further inspection of the class GLMPane, we can see which other messages are sent to these elements.

```
9  GLMPane>>resetAnnouncer
10         super resetAnnouncer.
11         self presentations do:
12                 [ :each | each resetAnnouncer]
13
14 GLMPane>>update
15         ...
16         self presentations do: [ :each | each update ]
```

Listing 2. Access to the elements of the variable presentations

In Listing 2 we see in lines 12 and 16 that resetAnnouncer and update are sent to elements of presentations as well.

[3] In Smalltalk terminology, to invoke a service of an object, one "sends it a message". A message consists of a "selector" (the name of the message) and the arguments. The receiver is then free to decide which "method" to use to respond to that message.

[4] The term "Pharo image" is used to denote snapshot of the running Pharo system, frozen in time.

[5] The system used with this example is Moose 5.0, a platform for software and data analysis based on Pharo. The actual number of implementations may vary in the other systems.

After taking this information into account, there are still 63 possible types for the elements of the variable each. (Bear in mind that this kind of analysis requires control and flow analysis, thus is more difficult to perform automatically.)

```
GLMPane>>addPresentations: aCollection                    17
    self notingPresentationChangeDo:                      18
        [ aCollection do: [ :each |                       19
            self addPresentationSilently: each ]          20
        ]                                                 21
                                                          22
GLMPresentation>>pane                                     23
    ^ pane ifNil: [                                       24
        pane := (GLMPane named: 'root'                    25
                    in: GLMNoBrowser new)                 26
            addPresentationSilently: self;                27
            yourself]                                     28
```

Listing 3. Senders of the method addPresentations:

Analysing senders of addPresentationSilently: does not yield much insight into the type of the argument each. In lines 20 and 26 in Listing 3 we can see that the method is invoked twice within the source code. Line 20 provides no information about the elements each of the method argument aCollection. In line 27, the argument passed is self, which is an object either of type GLMPresentation or any of its subtypes, which leaves us again with 63 possible types for the method argument in line 6 in Listing 1. As in the previous explanation, this kind of analysis depends on control-flow, hence is difficult to perform automatically in dynamically typed languages [38].

One can reasonably suppose that the name of the instance variable presentations reveals the type of its elements, since the class GLMPresentation is present in the project. In this case the name is not of much help, since the class GLMPresentation has 63 subclasses.

We propose to exploit information about the usage of the classes representing possible types of the variable each to highlight which classes are more likely to represent the variable's actual type(s) at run time. We argue that the degree of usage of the classes throughout the source code is strongly related to the likelihood of the class being used as a type for a particular variable. We have developed a couple of heuristics based on the different ways of class usage in an aim to improve type inference techniques.

One of the heuristics sorts the possible types based on the number of occurrences of the name of the class throughout the image. In the example from the Listing 1, the list of possible types for the argument each contains 75 classes in the following order, sorted by the frequency of the class names occurring in the image:

1. GLMTabulator

2. GLMCompositePresentation

3. GLMFinder

4. GLMPaneAdded

5. MooseFinder

...

75. PaneAbstractLine

As a result, a developer will be issued with the above presented list of 75 possible types in the specified order. The actual type of the variable at run time is GLMTabulator in multiple runs of the system, which is at the top of the list. We argue that it is possible to introduce accurate information about the type of a variable with such heuristics, avoiding "false positives", and that this will provide more insightful information to developers for program comprehension.

3. Related Work

The pioneer in this field was Milner, who developed a polymorphic type inference algorithm named "Algorithm W" [27]. It was first implemented as a part of the type system of the programming language ML. Milner's original algorithm addresses universal polymorphism (*i.e.*, genericity) but not subtype polymorphism."Algorithm W" is developed for simple applicative language Exp, where types are considered to be hierarchies of functional types over a set of basic types.

Some type inference approaches are founded on the fact that types can be seen as sets of expressions, instead of sets of values. One such has been developed for functional language FL to be used for the purpose of optimisation [3], which requires for type inference to be as precise as possible.

RoelTyper is a fast and relatively accurate type inference technique [31]. Types inferred based on the variable's interface are then merged with the assigned types, in ways such that priority can be given either to the interface types or to the assigned types. This approach was used as a basis for our prototype implementation as well as for the prototype implementation of the type inference algorithm named EATI, which tracks the frequency of sending a message to the instance of the particular class throughout the language ecosystem [34]. This frequency is then used to augment the type information. This approach showed a significant improvement when compared to its *RoelTyper* implementation, thus we used it as a measuring stick.

One of the recognised type inference algorithms is the Cartesian Product Algorithm [1], also known as CPA, originally developed for *Self*, a dynamically typed language [41]. This algorithm benefits from the observation that the return type of a method may depend on the types of the method arguments. CPA is used in the Starkiller type inference engine [32], a type inferencer and compiler for Python. One of the extensions of the CPA algorithm is DCPA [42], which considers also *data polymorphism*, *i.e.*, when a variable can have assigned values of different types.

Alexander Spoon *et al.* have developed one of the most precise type inference algorithms, named Chuck [36, 37], a *demand-driven* algorithm using *subgoal pruning*. They

calculate initial tables before type inference is performed and incrementally update them.

Execution of a program is also used to gather types. This approach has been used by Jong-hoon *et al.* to infer types for Ruby [11]. Wrappers around variables are used to collect the constraints during run time. These constraints are later used to infer types. The main obstacle to this kind of approach is that it requires runnable source code with valid input, either through test cases or symbolic execution [14].

Another related field of work is optional typing. This kind of type system frees the developer from types in general, but leaves him the opportunity to specify the type when deemed necessary [4]. Strongtalk is an optional type system developed for Smalltalk [20]. Pegon [33] is also an optional type system for Smalltalk, inspired by Strongtalk, developed recently. A pluggable type system takes optional types one step further, and allows a language to support multiple, optional type systems. TypePlug is an implementation of this idea for Smalltalk, exploiting annotations to encode types [21]. Another example of an optional type system is DRuby [17]. DRuby uses both static and dynamic information. Developers can annotate their code and these annotations will be verified using run-time information.

There is also the field of gradual typing, which allows some variables to have types specified at compile time, and other variables may be left untyped. Correctness of typed variables is checked at compile time, and the type system ensures that these types are not violated during a program run. Typed Racket [40] is an extension to Racket programming languages, which aims to support statically typed Racket programs [39]. The main idea is to support existing language idioms. There is a gradual type system for Smalltalk, named Gradualtalk, developed by Allende *et al.* [4] and inspired by Typed Racket.

4. Heuristics and Approaches

4.1 Terminology

To explain the heuristics, we introduce a simple set-theoretic model in Figure 1 that captures key properties for the entities shown in the UML diagram in the Figure 2. We have also previously presented this algorithm [28].

$$msg : V \to \mathcal{P}(S) \tag{1}$$

$$sel : M \to S \tag{2}$$

$$def_m : M \to C \tag{3}$$

$$sup : C \to C \cup \{null\} \tag{4}$$

$$assign\_types : V \to \mathcal{P}(C) \tag{5}$$

$$under : C \cup \{null\} \times S \to \{true, false\} \tag{6}$$

Figure 1. The core model.

Given a target program language, C is the domain of all classes, M is the domain of all methods, S is the domain of all selectors. V is the domain of all variables, including instance variables, method arguments and local variables.

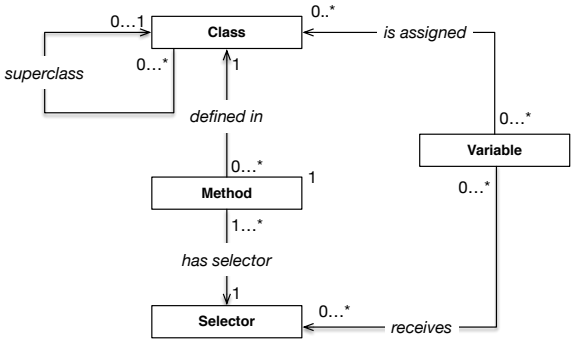

Figure 2. The core model in UML

Each variable v has a (possibly empty) set of messages $msg(v)$ sent to it in its lexical scope (1) either directly or through the getter methods. Note that in this paper we consider the lexical scope for instance variables only to be the methods of the class in which it is defined, but not its subclasses. In case of instance variables inherited from superclass, we choose to treat them as instance variables defined in the subclass. The implications of this decision are discussed in section 6. We call this set of messages the interface of the variable v. Each method m has a unique selector $s = sel(m)$ (2), and is defined in a unique class $c = def_m(m)$ (3). Each class c has a unique superclass $c' = sup(c)$ (4). We define the superclass of Object to be *null*, *i.e.*, *sup*(Object) = *null*.

Consider the example class hierarchy in Figure 3. In this example we see a class RTGlobalBuilder with an instance variable named properties and methods addProperty:, execute and initialize. Within these three methods messages sent to the instance variable properties are

$$msg(\mathsf{properties}) = \{\mathsf{add:}, \mathsf{do:}\}$$

Also, each variable v may have one or more assigned types $c \in assign\_types(v)$ (5) if the variable v is the left side of an assignment where the right side of the same assignment is a message send to a class which results in creating a new object, *i.e.*, is a call to a constructor or this newly created object has been assigned to the variable via setter methods. Returning to the example, in the method RTGlobalBuilder>>initialize there is an assignment to the instance variable properties of the newly created object of type OrderedCollection, which means that

$$assign\_types(\mathsf{properties}) = \{\mathsf{OrderedCollection}\}$$

We have used a couple of heuristics to guess the type of the expression result assigned to the variable, as done with RoelTyper [31]. These heuristics are listed in the Table 1.

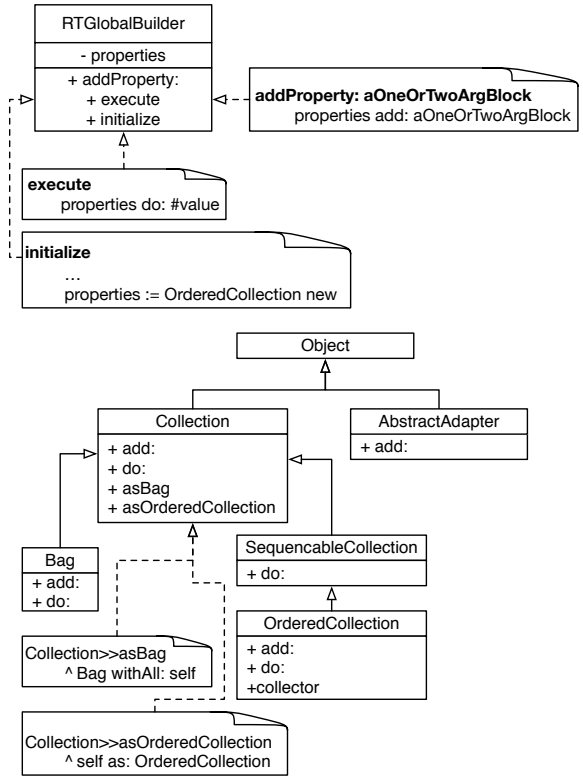

Figure 3. Sample class hierarchy

Having multiple assignments to the same variable is possible, but it is beyond the scope of our small example.

$$\mathit{under}(c, s) = s \in \mathit{sel}(\mathit{def}_m^{-1}(c)) \vee \mathit{under}(\mathit{sup}(c), s) \quad (7)$$

$$\mathit{intr}(c) = \{s \in S | \mathit{under}(c, s) = true\} \quad (8)$$

$$\mathit{sel\_types}(v) = \{c \in C | \mathit{msg}(v) \subseteq \mathit{intr}(c)\} \quad (9)$$

Figure 4. Computing possible types for a variable.

We can now query the model to ascertain the set of possible types for every variable. Each class c can either understand the selector s or not (6). The class c understands selector s if this class defines a method $m \in \mathit{def}_m^{-1}(c)$ such that $\mathit{sel}(m) = s$ or its superclass $\mathit{sup}(c)$ understands it (7), as presented in Figure 4. We also define that *under*($null$, s) = $false$. The interface of the class c is a set $\mathit{intr}(c)$ of all the selectors s that class c understands (8). The class c is a possible type for the variable v if class c understands the interface of the variable v. (9).

In the example in Figure 3 we see that the class Collection understands the selector add:, as do all of its subclasses and the class AbstractAdapter, while *under*(RTGlobalBuilder, add:) = $false$.

We can now calculate the interfaces of the classes

$$\mathit{intr}(\text{Collection}) = \mathit{intr}(\text{Bag}) = \mathit{intr}(\text{SequenceableCollection}) = \{\text{add:}, \text{do:}, \text{asBag}, \text{asOrderedCollection}\}$$

$$\mathit{intr}(\text{OrderedCollection}) = \{\text{add:}, \text{do:}, \text{collector}, \text{asBag}, \text{asOrderedCollection}\}$$

$$\mathit{intr}(\text{AbstractAdapter}) = \{\text{add:}\}$$

From the previous equations, we see that

$$\mathit{msg}(\text{properties}) \subseteq \mathit{intr}(\text{Collection})$$
$$\mathit{msg}(\text{properties}) \subseteq \mathit{intr}(\text{Bag})$$
$$\mathit{msg}(\text{properties}) \subseteq \mathit{intr}(\text{SequenceableCollection})$$
$$\mathit{msg}(\text{properties}) \subseteq \mathit{intr}(\text{OrderedCollection})$$
$$\mathit{msg}(\text{properties}) \not\subseteq \mathit{intr}(\text{AbstractAdapter})$$

Hence, possible types for the variable properties are

$$\mathit{sel\_types}(\text{properties}) = \{\text{Collection}, \text{Bag}, \text{SequenceableCollection}, \text{OrderedCollection}\}$$

4.2 Heuristics

The first intuition was that the classes used most throughout the source code are more "present" in the source code, and are more likely to represent the types of the variable than classes which are less "present" in the source code. Classes can be used "on their own", as independent objects, or they can be used to instantiate new objects. Hence, we have implemented and evaluated two possible heuristics:

1. Name occurrence heuristic
2. Class instantiation heuristic

We continue by explaining the evaluated heuristics used for ordering the possible types of a variable v. We use each heuristic to order separately two sets of classes: *assign_types*(v) and *sel_types*(v).

4.2.1 Name Occurrence

The intuition is that the more the name of a class is used throughout the source code, the more it is likely that that class represents the type of a variable. This heuristic is founded on the calculation of the occurrences of class name throughout the source code. We count these occurrences and use the relative frequency to sort the possible types of a variable.

In our example in Figure 3, we encounter two occurrences of the class OrderedCollection: in the methods Collection >>asOrderedCollection and RTGlobalBuilder>>initialize. Hence,

$$\mathit{name\_occ}(\text{OrderedCollection}) = 2$$

The class Bag is used as a receiver for a message send withAll: in the method Collection>>asBag, hence

$$\mathit{name\_occ}(\text{Bag}) = 1$$

Expression	Inferred type
$x = y \quad x == y \quad x < y \quad x > y \quad x <= y \quad x >= y \quad x \ = y$	Boolean
$x\ msg\ y$, where msg is any of the arithmetic, logarithmic or trigonometric functions or functions used to round a number	Number

Table 1. Heuristics used to infer the type of the expression

Classes Collection and SequenceableCollection are mentioned nowhere in the code, except for its declaration, which we do not count, so

name_occ(Collection) = 0

name_occ(SequenceableCollection) = 0

Based on the obtained information, we can now sort the possible types for the variable properties. As we have mentioned at the beginning of subsection 4.2 for each variable v we sort independently the lists *assign_types*(v) and *sel_types*(v).

In the example, the list *assign_types*(properties) has only one element, so there is no need for sorting. But the list *sel_types*(properties) has four elements which will be sorted as follows:

1. OrderedCollection
2. Bag
3. Collection
4. SequenceableCollection

4.2.2 Class Instantiation

Since a class can be used as an object itself, *e.g.*, to invoke a class method which does not necessarily need to create a new object, in this heuristic we are focusing only on the places in source code where a class name is used to instantiate a new object. In Smalltalk the usual way to create a new object is to send a message new to a class. Besides the methods with the selector new, all the methods which belong to any of the protocols[6] "initialize", "initialization" and "instance creation" are considered to be constructors, *i.e.*, result in a new object creation. So, we count all the occurrences of class usage like OrderedCollection new or any other message send to a class which results in invoking a method from any of the previously mentioned protocols.

Returning to the example in Figure 3, the only place in the source code where a class is instantiated is within the method RTGlobalBuilder>>initialize, where a new object of the type OrderedCollection is created and assigned to the instance variable properties. Based on this information, we can count the corresponding values for each class:

class_inst(Collection) = *class_inst*(Bag) = *class_inst*(SequenceableCollection) = 0

class_inst(OrderedCollection) = 1

As in the previous heuristic, there is only one assigned type to the variable properties, so there is no need for sorting the list *assign_types*(properties). However, the sorted list *sel_types*(properties) will look like:

1. OrderedCollection
2. Bag
3. Collection
4. SequenceableCollection

4.3 Assigned Types vs. Selector Types

In our evaluation we give the priority to the assigned type, since we believe that this information is inserted by a developer in a high accuracy. As a result of the inference process, for each variable v two lists will be presented, namely, *assign_types*(v) and *sel_types*(v), so that a developer has a notion of types explicitly being assigned to the variable, and the types that are implicitly determined. If there are no assignment types for a variable v, only selector types are presented.

4.4 Approaches

In our opinion, it is important for the developer to know the specific type of a variable, but also to have a notion of the hierarchy of classes to which the run-time type of the variable can belong. For that reason, we have developed both a hierarchy-based approach and a class-based one. To the best of our knowledge, this is one of the most simple algorithms that tries to infer the precise type for variables, and not just the class hierarchy. As the tool is intended purely to assist developers in program comprehension, we consider it important to infer both possible classes and hierarchies to which the type of the variable may belong.

For each variable v both approaches collect the messages sent to the variable v. The main difference thereafter is in computing the set of classes which understand the interface of the variable, *sel_types*(v).

4.4.1 Class-Based Approach

In this approach, we try to infer the specific class which represents the type of a variable. This approach is explained in the example in Figure 3 throughout subsection 4.1. Hence,

[6] Methods in Smalltalk are organized in protocols, *i.e.*, groups of related methods.

the set *sel_types*(v) will contain all the classes in the image that understand the interface of the variable v. In the example in Figure 3, the sets of possible types for the variable properties are

$$\textit{assign\_types}(\text{properties}) = \{\text{OrderedCollection}\}$$

$$\textit{sel\_types}(\text{properties}) = \{\text{Collection, Bag, SequenceableCollection, OrderedCollection}\}$$

so, for each of the proposed heuristics, we proceed by sorting these classes based on the heuristic's rules.

4.4.2 Hierarchy-Based Approach

A variable can have an interface understandable by tens, hundreds, or even thousands of classes. Obviously, such information presented to a developer in a case like this is not helpful. Our idea is to identify the highest class in a hierarchy of classes that understands the interface of the variable. Throughout the evaluation process, most of the variables had an interface understandable by tens of independent hierarchies, *i.e.*, hierarchies whose root classes do not have a common superclass understanding the same interface. In this case the inferred types of the variable would be the root classes of the identified hierarchies. Let us remember that in the example in Figure 3 the interface of the instance variable properties was understandable by four classes: Bag, Collection, OrderedCollection and SequenceableCollection. All four classes belong to the same hierarchy of classes having the class Collection as its root class. In the hierarchy-based approach, we will infer the type of the variable properties as

$$\textit{sel\_types}(\text{properties}) = \{\text{Collection}\}$$

Sets of messages sent to a variable can be understandable by multiple hierarchies, but this is beyond the scope of our small example.

Let us emphasise here that no change is made to the set *assign_types*(v), but only to the set *sel_types*(v). We consider the set of explicitly assigned types to a variable to be truthful, as it is. The implications of this decision are discussed in section 6.

5. Evaluation

We have implemented a proof-of-concept for Pharo, a dialect of Smalltalk and a highly reflective dynamically typed language, which allows fast and easy implementation of analysis tools [16].

In order to evaluate our assumptions, we have used five open-source projects, written in Pharo, for which we were able to collect run-time information that closely depicts their real usage: Roassal2[7] [5], Glamour[8] [7], Bloc[9], Morphic[15] and Moose[10] [12, 13, 19]. Roassal is an agile visualisation engine which graphically renders objects using short and expressive Smalltalk expressions (*i.e.*, an internal DSL). Glamour is a framework to describe the navigation flow of browsers. Bloc project is a redesign of Morphic, a User Interface construction kit. Moose is a platform for software analysis. We wanted to avoid the usage of unit tests, because there is no guarantee that they will reflect the complete picture of the project's usage. Four of these projects (Roassal, Glamour, Morphic and Bloc) provide a number of "example methods" which reflect the real usage of the corresponding project: Roassal2 has 948, Glamour 68, Morphic 29 and Bloc 202 of these methods. They were run and run-time information about types of variables, *i.e.*, classes that represent the type of the object stored in the variable, during the execution of these methods was recorded. Run-time data for the Moose project was collected by actually performing software analysis on a couple of projects.

By instrumenting the source code of the projects to log the types of the variables, including instance variables, method and block arguments, and method and block temporary variables, and running these examples, we have recovered the run-time types of the variables. For this purpose we have used a mechanism to track the types about variables during run time, built on top of Reflectivity[11], a tool used to annotate AST nodes with metalinks. We consider these types to be the actual, real types of the variables during run time, and hold those types as truthful information to which results provided by the heuristics are compared.

Types of these variables are then inferred using the two heuristics. We have statically and dynamically collected enough information to infer the types of 5246 variables in these five projects and evaluate the results. This means that at least one message is sent to a variable, or at least one assignment of the newly created object to a variable is encountered in the source code during static analysis and that we had access to the run-time information about the types of these variables. Examples that we used to collect the run-time information about types covered 6009 variables in Roassal2, 259 variables in Glamour, 1006 variables in Morphic, 220 variables in Bloc and 378 variables in Moose, thus in total 7872 variables. For 2626 variables, there was not enough information to infer types statically, *i.e.*, there was no assignment to the variable, nor any message sent.

The evaluation section is divided into two parts, each focusing on one of the used heuristics. We try to answer the following research questions:

1. How well does each of the heuristics work?
2. For what kind of types, *e.g.*, library or project-related types, are the results correct?

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

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

[9] http://www.smalltalkhub.com/#!/~AlainPlantec/Bloc

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

[11] http://www.smalltalkhub.com/#!/~RMoD/Reflectivity

3. How useful is it to infer the hierarchy of the possible variable type?

5.1 Name Occurrence Heuristic

A summary of the evaluation results is given in Table 2.

5.1.1 Guessed and "Near-Guessed" Types

For 563 analyzed variables, our algorithm was not able to provide any information about the assigned types, and the messages sent to these variables were defined in class Object , thus there was only enough static information to conclude the possible type as Object. We argue that these results could be discarded since they are easily identifiable and they do not provide any useful information to the developer. Beside 441 already defined methods in the Object class in Pharo, it is also possible to add a user-defined method in library classes. These messages can be sent to any Smalltalk object. A messages commonly sent to these 563 variables are:

- rtValue: which is specific to the Roassal2 project, but is defined as an extension method in the Object class and behaves the same as the message value, *i.e.*, returns the object to which the message is sent.
- comparison messages, like $=$, $==$, $!=$ *etc.*
- message value whose implementation is in Object class
- message at: which assumes that the receiver is indexable and answers the value of an indexable element in the receiver
- messages which check whether the object is *null*, *e.g.*, ifNil:, ifNotNil: *etc.*

The results show that in about 51.57% of cases (for 2415 out of the remaining 4683 variables in the five projects) by using this simple heuristic we can successfully infer the *correct classes* that represent the types of the variable. We emphasize that the type inference heuristics are inferring the *correct classes*, since similar approaches developed for dynamically typed languages, *e.g.*, *RoelTypes*[31] and *EATI*[34] infer solely the hierarchy of classes that represent the potential type of the variable. By analysis of the inferred types, we have established that this heuristic is working well both for project-related types, as well as for the types from the standard library, since 1179 of the correctly inferred types are project-related, and 1236 are library types.

If the variable has n run-time types, where $n > 1$, we consider it as "guessed" if the set of first n types of the statically inferred list of types is the same as the set of run-time types. Thus, we also count variables that are considered to be duck-typed.

In the same table we can see that for 536 additional variables, *i.e.*, 11.45% of variables, the heuristic failed to promote the correct type at the top of the list, but the correct type is present in the top three. We call them "near-guessed". If the variable has n run-time types, where $n > 1$, we consider it as "near-guessed" if the set of first $n + 2$ types of the statically inferred list of types is the superset of the set of run-time types. For example, if the variable has three different run-time types, we consider it as "near-guessed" if all three types are present in the top five types in statically inferred list.

5.1.2 Incorrectly-Guessed Types

There are 1732 (36.98%) additional variables for which the correct type was not in the top three types in the list.

After closely analysing these variables, we have discovered that 74 of these variables, *i.e.*, 4.27% had a run-time type UndefinedObject which represents the *null* object in Smalltalk. Eleven of these variables had only *null* assigned during the program run. Depending on the language syntax these variables could be left out of the evaluation or not, since their run-time type is only *null*. In Java, where *null* is literal, these variables should be discarded. In Smalltalk *null* is not a literal but rather the sole instance of the class UndefinedObject (subclass of the class Object). We therefore choose to keep them. When considering statically inferred types, we have discovered 26 variables with an assigned type of UndefinedObject, and an interface consisting exclusively of messages defined in the class Object . Thus, we can only infer their types as sets of classes UndefinedObject and Object. Again, we choose to preserve these variables, since UndefinedObject is a valid class in Smalltalk.

We have analyzed sets of messages sent to these variables, and discovered that 190 of these variables had received only type predicates, *e.g.*, isCircle and conversion methods, *e.g.*, asOrderedCollection, thus their employment in type inference algorithms could result in small improvements. We have also discovered that 54 variables have the interface composed solely of boolean predicates, which may imply that these variables are of type Boolean. This presents an opportunity for improvement. Most of these combinations of messages (86.35%) have been sent to a maximum of three variables.

Most of the remaining variables, *i.e.*, 1570 out of 1658 do not have any assigned type, hence inferring their type depends solely on their interface. Half of these variables have an interface understood by more that 323 classes and their type can belong to more than nine different class hierarchies *i.e.*, classes which do not have a common superclass understanding the required set of messages. That is why we feel it appropriate to infer not only the specific type of the variable, but also the hierarchy to which it may belong.

Interestingly, 114 of the incorrectly-guessed variables have been assigned in the source code multiple newly created objects whose types do not correspond to the actual run-time type of the variable.

5.1.3 Hierarchy-Based Approach

Figure 5 shows the distribution of the number of possible hierarchies for each statically analysed variable. More than

Class-based approach							
Project name	#of analyzed variables	#of guessed variables	#of guessed variables - library type	#of guessed variables - package type	#of near-guessed variables	#of incorrectly-guessed variables	#of Object type
Roassal	4143	2026 (53.87%)	945	1081	423 (11.25%)	1312 (34.88%)	382
Bloc	220	95 (46.57%)	75	20	31 (15.2%)	78 (38.23%)	16
Glamour	136	53 (51.96%)	32	21	5 (4.9%)	44 (43.14%)	34
Moose	150	48 (40%)	38	10	18 (15%)	54 (45%)	30
Morphic	597	193 (38.91%)	146	47	59 (11.89%)	244 (49.19%)	101

Hierarchy-based approach							
Project name	#of analyzed variables	#of guessed variables	#of guessed variables - library type	#of guessed variables - package type	#of near-guessed variables	#of incorrectly-guessed variables	#of Object type
Roassal	4143	2655 (70.59%)	1464	1191	623 (16.56%)	483 (12.84%)	382
Bloc	220	114 (55.88%)	87	27	55 (26.96%)	35 (17.15%)	16
Glamour	136	84 (82.35%)	40	44	7 (6.86%)	11 (10.78%)	34
Moose	150	73 (60.83%)	57	16	9 (7.5%)	38 (31.67%)	30
Morphic	597	373 (75.20%)	204	169	48 (9.67%)	75 (15.12%)	101

Table 2. Name occurence heuristic

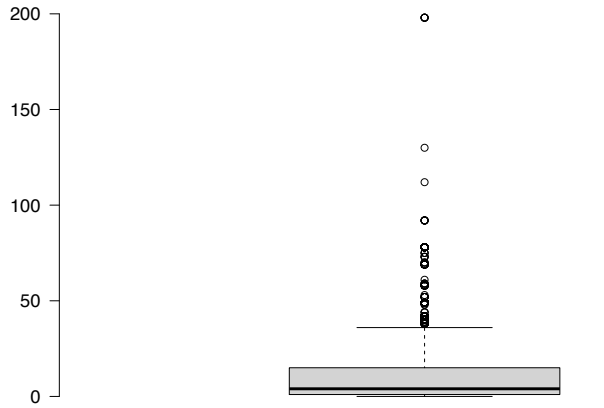

Figure 5. Distribution of the number of possible hierarchies for each statically analysed variable

half of the variables (55%) have an interface understandable by more than 4 independent hierarchies, and at least 25% of the variables have a possible run-time type which may belong to more than 14 hierarchies. That is why we deem it necessary to infer the possible hierarchy for a variable.

When we apply the hierarchy-based approach, we can see that for 3299 variables (70.44%), we are able to correctly infer the hierarchy of possible types for the variable. The corresponding results, labeled *hierarchy-based approach*, can be seen also in Table 2. We again discard the variables for which we have not been able to infer any other type but Object. For 15.84% of variables, *i.e.*, 742 variables the correct hierarchy was not at the top of the list, but present in the top three.

Both RoelTyper [31] and EATI[34] employ a *hierarchy-based* approach for inferring types. Since EATI showed a twofold improvement when compared to its basis, *i.e.*, RoelTyper, we have decided to compare our results with the results produced by EATI. The comparison is explained in Section 5.3.

Class-based approach							
Project name	#of analyzed variables	#of guessed variables	#of guessed variables - library type	#of guessed variables - package type	#of near-guessed variables	#of incorrectly-guessed variables	#of Object type
Roassal	4143	2008 (53.39%)	963	1045	362 (9.62%)	1391 (36.98%)	382
Bloc	220	87 (42.64%)	67	20	32 (15.69%)	85 (41.67%)	16
Glamour	136	52 (50.98%)	31	21	5 (4.9%)	45 (44.11%)	34
Moose	150	45 (37.5%)	36	9	17 (14.17%)	58 (48.33%)	30
Morphic	597	181 (36.49%)	136	45	62 (12.5%)	253 (51%)	101

Hierarchy-based approach							
Project name	#of analyzed variables	#of guessed variables	#of guessed variables - library type	#of guessed variables - package type	#of near-guessed variables	#of incorrectly-guessed variables	#of Object type
Roassal	4143	2584 (68.7%)	1429	1155	557 (14.8%)	620 (16.48%)	382
Bloc	220	105 (51.47%)	78	27	60 (29.41%)	39 (19.11%)	16
Glamour	136	84 (82.35%)	39	45	6 (5.88%)	12 (11.76%)	34
Moose	150	68 (56.67%)	54	14	8 (6.67%)	44 (36.67%)	30
Morphic	597	364 (73.39%)	193	171	55 (11.1%)	77 (15.52%)	101

Table 3. Class instantiation heuristic

5.2 Class Instantiation Heuristic

5.2.1 Guessed and Near-Guessed Types

The results are similar to the previous heuristic *i.e.*, in 50.67% of cases (for 2373 out of 4683 variables in all the projects) we succeeded to correctly infer the actual class that represents the type of the variable, as shown in Table 3. This heuristic is also working well both for project-related types, as for the types from the standard library: 1140 of the correctly inferred types are project-related, and 1233 are library types.

Another 478 variables, *i.e.*, 10.2% of variables, are "near-guessed".

5.2.2 Incorrectly-Guessed Types

For 1832 variables the correct type was not in the top three types in the list.

UndefinedObject was present as a run-time type for 75 of these variables. For 26 variables we did not have enough information to conclude anything more than types UndefinedObject or Object. These types can be discarded, depending on the language, as already discussed.

These variables have the interfaces understood by tens or hundreds of classes: at least half of the variables could have more than 301 different types. Most of them have an interface which is understandable by multiple hierarchies. The number of these hierarchies is ranging from 2 to 203 per variable, with a median of 11.

There are 115 incorrectly-guessed variables that have been assigned types not corresponding to their run-time types. For example, one of the variables has the assigned object of type OrderedCollection, while the run-time types are Array and OrderedCollection. Since both classes are subclasses of SequenceableCollection class, the hierarchy-based approach will prove itself useful in this situation.

5.2.3 Hierarchy-Based Approach

In the hierarchy-based approach, for 3205, *i.e.*, 68.44% variables, we are able to correctly infer the hierarchy of possible

Type inference	#of analyzed variables	#of guessed variables	#of near-guessed variables	#of Object variables
Name occurrence heuristic — hierarchy-based	5246	3299	686	563
EATI	5246	2080	748	961

Table 4. Comparison with EATI

types. This heuristic is working slightly worse than "name occurrence" heuristic. For 792 variables, *i.e.*, 16.91%, correct run-time types were "nearly-guessed".

5.3 Comparison with EATI

Since the "name occurrence" heuristic has shown better results among the implemented heuristics, we compare its *hierarchy-based approach* to EATI, a type inference technique that uses information available in the language ecosystem [34], in contrast to approaches that use only information available in the project. It has a simpler version of fast type inference technique presented by Pluquet [31] as the basis for its prototype implementation. It collects only messages sent directly to a variable, and not those sent through "getter" methods, and likewise for the assignments and "setter" methods. EATI gathers data about the frequency of message sends to instances of available types from the software ecosystem, and stores it in a central repository, for future queries. When possible types for a variable have been inferred, based on the assignments and message sends, the likelihood of the variable being of the actual type "is computed based on how many times the messages sent to this variable have been observed to be sent to each potential type throughout the ecosystem" [34]. The information produced by EATI is the list of highest possible classes in the hierarchy that understands the set of messages sent to a variable. That is why we compare it with our *hierarchy-based approach*. EATI shows a twofold improvement when compared to a single system type inference techniques, *i.e.*, RoelTyper [31] which also infers hierarchy roots as possible types for the variable.

The results of our comparison are shown in Table 4. We have calculated the number of variables for which EATI succeeded to promote the correct type of the variable to the top of the list, and compared it with the number of variables for which "name-occurrence" heuristic also succeeded to promote the correct type to the top of the list (#of guessed variables). We have also compared the number of variables for which both of the inference approaches failed to promote the correct type to the top of the list, but the correct type was among the first three on the list (#of near-guessed variables). On the same set of variables, the "name-occurrence" heuristic performed significantly better: it correctly infers 58.6% more variables. Even if we count also "near-guessed" together with "guessed" (since they can be considered as reasonable results to present to developer) we can see that there is an improvement of 40.9%. We deem these findings important, since our heuristic yields better results, and they are obtained with considerably less effort and resources. It should be noted also that EATI does not work well for the project-related types as it lacks awareness of these types, and concentrates more on the types available broadly in the ecosystem, while our heuristics produce reasonable results both for project-related and library types.

As can be seen in the same table, we have left out the variables for which the two techniques were not able to produce any other possible type than Object. As well, in this section our algorithm showed an improvement of 41.4%, as it was able to collect more information about variables due to the slight changes of the manner of data gathering.

6. Discussion and Threats to Validity

Even though these heuristics should work on every dynamically typed language, we cannot state this without more study.

The main threat to validity comes from the dynamic information we have used to evaluate our heuristics. We have chosen the projects *Roassal2*, *Glamour*, *Bloc*, *Morphic* and *Moose* to evaluate the heuristics, since these projects benefit from sets of realistic example methods. We have chosen these examples over the unit tests, since we feel they characterize the real usage of the projects, thus providing more insight into the actual software behaviour. It is an open question whether this set of examples is truly representative of executions of these projects.

Another threat arises from the use of dynamic features and type predicates in Smalltalk. Dynamic features are seldom used in Smalltalk, but they are used to the extent that they must be taken into consideration [8], especially dynamic message send. Type predicates are prevalently used in Smalltalk [9]. In the manual investigation of our results, we have encountered variables queried for their type by sending the message isKindOf: and then being treated differently based on the answer. Since type inference heuristics presented in the paper are intended to be fast, they are flow-insensitive.

Our consideration of the lexical scope of an instance variable may have influenced the results. By all means, considering all the methods in the class defining the instance variable, along with the methods of the subclasses, as the lexical scope of the variable may only improve the results.

We chose to treat the assignment types of a variable to be truthful as they are, without considering the subtypes. If the assigned type is Collection, we do not consider as the possible type of the variable Bag, OrderedCollection, nor

SequenceableCollection (Figure 3). This choice is the same in both class-based and hierarchy-based approach. If we would consider the assigned types along with all subtypes in class-hierarchy approach, the number of correctly inferred types may increase, due to the increased number of possible types for the variable.

While our first idea was to explore "name occurrence" heuristic and "class instantiation" heuristic for type inference in dynamically typed languages, we have also tried to evaluate the heuristic that sorts the classes available in the image based on their number of instances live in the image. Since Pharo is a highly reflective and interactive IDE that supports live programming, many classes may have live instances within the image, but this heuristic showed to be the least precise. The problem with this approach was that only 744 out of 8321 classes have live instances, compared to 4066 classes whose name is mentioned in the image, or 2906 classes that are instantiated somewhere in the image. This also can explain why the "name occurrence" heuristic performs slightly better than the "class instantiation" heuristic.

We intend to improve on the "class instantiation" heuristic by analysing the ways a developer can create a new object of a class. In this paper, we were only intercepting when the messages that belong to any of the protocols initialize, initialization and instance−creation are sent to a class. Smalltalk is a highly reflective language, which includes dozens of ways to create a new object, and also allows a developer to define new ways by implementing class side methods.

One of the problems is also the set of available classes for every variable. Since Smalltalk is a dynamically typed language without a main method, determining the set of available classes for every variable requires control and flow analysis, which is in general an NP-hard problem [38]. We have tried to apply the approach used to guess the type of the method arguments in Smalltalk [35], but the results were on average 30% worse than without applying this approach.

During our evaluation, we have statically analysed 9732 variables and collected dynamic information to evaluate types of 7872 variables. The intersection of these two sets yielded 5246 variables. The reason for this is that for 2626 variables for which we had run-time data there was not enough static information to infer their type, *i.e.*, no message had been sent to the variables, and they had no assigned type, and for 4486 variables for which we had static information, there was no available dynamic information about their type. 7099 of these variables receive messages understandable by more than one hierarchy, and 49 of them have an interface not understood by any single hierarchy ,*i.e.*, any single class in the image. We suppose that those 49 variables are so-called "duck-typed" variables, and that 7099 variables have the possibility of being duck-typed. That is why we deem important for future work to explore the actual usage of duck-typed variables, for these variables can be considered as pollution to any type-inference algorithm. Since more than 72.94% of variables can belong to more than one hierarchy, we consider necessary the *hierarchy-based* approach we have implemented, in order to provide a notion to a developer of what the type of the variable can be.

7. Conclusion and Future Work

Static type information comes in handy when performing program comprehension tasks in dynamically typed languages. Current type inference algorithms tend to become very complex in order to provide accurate information, and still they suffer from the problem of "false positives". We have presented a couple of cheap heuristics that aim to provide precise type information by using simple algorithms. The implemented prototype for Pharo Smalltalk allows us to assess the proposed heuristics.

Heuristics produced results comparable with the existing type inference algorithms, and tend to work quite well both for library and project-related types. While they can benefit from improvements, even in their simple form they provide us with promising results. Hierarchy-based heuristics are working better than class-based ones, which is to be expected. One reason for this is that more than 72.94% of analysed variables have an interface understandable by more than one hierarchy.

During the analysis of the proposed heuristics, several opportunities for improvement have been remarked. With more complex approaches, it would be possible to identify the set of available classes for the project, so that the set of possible types of a variable can be shortened. Lexical similarities in between the variable name and class name may reveal the possible type of the variable. Type predicates used to ask the variable for its type can also give a hint about the possible type and improve heuristics. We intend to explore these directions.

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).

References

[1] O. Agesen. The Cartesian product algorithm. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 2–26, Aarhus, Denmark, Aug. 1995. Springer-Verlag.

[2] O. Agesen, J. Palsberg, and M. I. Schwartzbach. Type inference of SELF: Analysis of objects with dynamic and multiple inheritance. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 247–267, Kaiserslautern, Ger-

many, July 1993. Springer-Verlag. URL http://www.cs.purdue.edu/homes/palsberg/publications.html.

[3] 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*, POPL '91, pages 279–290, New York, NY, USA, 1991. ACM. ISBN 0-89791-419-8. doi: 10.1145/99583.99621. URL http://doi.acm.org/10.1145/99583.99621.

[4] E. Allende, O. Callaú, J. Fabry, É. Tanter, and M. Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, Aug. 2013. doi: 10.1016/j.scico.2013.06.006. URL http://hal.inria.fr/hal-00862815.

[5] V. P. Araya, A. Bergel, D. Cassou, S. Ducasse, and J. Laval. Agile visualization with Roassal. In *Deep Into Pharo*, pages 209–239. Square Bracket Associates, Sept. 2013. ISBN 978-3-9523341-6-4.

[6] B. Boehm and V. R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, Jan. 2001. ISSN 0018-9162. doi: 10.1109/2.962984. URL http://dx.doi.org/10.1109/2.962984.

[7] P. Bunge. Scripting browsers with Glamour. Master's thesis, University of Bern, Apr. 2009. URL http://scg.unibe.ch/archive/masters/Bung09a.pdf.

[8] O. Callaú, R. Robbes, É. Tanter, and D. Röthlisberger. How developers use the dynamic features of programming languages: The case of Smalltalk. In *Proceedings of the 8th working conference on Mining software repositories (MSR 2011)*, pages 23–32, New York, NY, USA, 2011. IEEE Computer Society. doi: 10.1145/1985441.1985448. URL http://scg.unibe.ch/archive/papers/Call11aDynamicFeaturesMSR2011.pdf.

[9] O. Callaú, R. Robbes, É. Tanter, D. Röthlisberger, and A. Bergel. On the use of type predicates in object-oriented software: The case of Smalltalk. In *Proceedings of the 10th ACM Dynamic Languages Symposium (DLS 2014)*, pages 135–146, Portland, OR, USA, 2014. ACM Press. doi: 10.1145/2661088.2661091. URL http://pleiad.dcc.uchile.cl/papers/2014/callauAl-dls2014.pdf.

[10] R. Cartwright and M. Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, New York, NY, USA, 1991. ACM. ISBN 0-89791-428-7. doi: 10.1145/113445.113469.

[11] J. Davies, D. M. Germán, M. W. Godfrey, and A. Hindle. Software bertillonage: Finding the provenance of an entity. In *MSR'11: Proceedings of the 8th International Working Conference on Mining Software Repositories*, pages 183–192, 2011. doi: doi.acm.org/10.1145/1985441.1985468.

[12] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. In *Proceedings of CoSET '00 (2nd International Symposium on Constructing Software Engineering Tools)*, June 2000. URL http://scg.unibe.ch/archive/papers/Duca00bMooseCoset.pdf.

[13] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005. ISBN 88-464-6396-X. URL http://scg.unibe.ch/archive/papers/Duca05aMooseBookChapter.pdf.

[14] H. Eertink and D. Wolz. Symbolic execution of LOTOS specifications. Memoranda Informatica 91-47, TIOS 91/016, University of Twente, May 1991.

[15] H. Fernandes and S. Stinckwich. Morphic, les interfaces utilisateurs selon squeak, Jan. 2007.

[16] B. Foote and R. E. Johnson. Reflective facilities in Smalltalk-80. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 327–336, Oct. 1989.

[17] M. Furr. *Combining Static and Dynamic Typing in Ruby*. PhD thesis, University of Maryland, 2009. URL https://www.cs.umd.edu/~jfoster/papers/thesis-furr.pdf.

[18] J. D. Gannon. An experimental evaluation of data type conventions. *Commun. ACM*, 20(8):584–595, Aug. 1977. ISSN 0001-0782. doi: 10.1145/359763.359800. URL http://doi.acm.org/10.1145/359763.359800.

[19] T. Gîrba. The Moose book, 2010. URL http://www.themoosebook.org/book.

[20] 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*, POPL '90, pages 136–150, New York, NY, USA, 1990. ACM. ISBN 0-89791-343-4. doi: 10.1145/96709.96722. URL http://doi.acm.org/10.1145/96709.96722.

[21] N. Haldimann, M. Denker, and O. Nierstrasz. Practical, pluggable types. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 183–204. ACM Digital Library, 2007. ISBN 978-1-60558-084-5. doi: 10.1145/1352678.1352690. URL http://scg.unibe.ch/archive/papers/Hald07b-Typeplug.pdf.

[22] S. Hanenberg. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. *SIGPLAN Not.*, 45(10):22–35, Oct. 2010. ISSN 0362-1340. doi: 10.1145/1932682.1869462. URL http://doi.acm.org/10.1145/1932682.1869462.

[23] 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 *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 153 –162, June 2012. doi: 10.1109/ICPC.2012.6240483.

[24] 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*, PLATEAU '14, pages 1–11, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2277-5. doi: 10.1145/2688204.2688212. URL http://doi.acm.org/10.1145/2688204.2688212.

[25] P. Lutz and T. W. F. A controlled experiment to assess the benefits of procedure argument type checking. *IEEE Trans. Softw. Eng.*, 24(4):302–312, Apr. 1998. ISSN 0098-5589. doi:

10.1109/32.677186. URL http://dx.doi.org/10.1109/32.677186.

[26] 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.*, 47 (10):683–702, Oct. 2012. ISSN 0362-1340. doi: 10.1145/2398857.2384666. URL http://doi.acm.org/10.1145/2398857.2384666.

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

[28] 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. URL http://scg.unibe.ch/archive/papers/Milo16a.pdf. To Appear.

[29] 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*, ICPC '15, pages 25–35, Piscataway, NJ, USA, 2015. IEEE Press. URL http://dl.acm.org/citation.cfm?id=2820282.2820289.

[30] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, Mar. 1998. doi: 10.1109/2.660187. URL http://www.cs.indiana.edu/classes/c102/read/Ousterhout.pdf.

[31] 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*, pages 69–78, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-769-1. doi: 10.1145/1640134.1640145.

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

[33] R. Smit. Pegon. https://sourceforge.net/projects/pegon/.

[34] 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*, Onward! '14, pages 133–142, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3210-1. doi: 10.1145/2661136.2661141. URL http://scg.unibe.ch/archive/papers/Spas14c.pdf.

[35] B. Spasojević, M. Lungu, and O. Nierstrasz. A case study on type hints in method argument names in Pharo Smalltalk projects. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 283–292, Mar. 2016. doi: 10.1109/SANER.2016.41. URL http://scg.unibe.ch/archive/papers/Spas16a.pdf.

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

[37] S. A. Spoon and O. Shivers. Dynamic data polyvariance using source-tagged classes. In R. Wuyts, editor, *Proceedings of the Dynamic Languages Symposium'05*, pages 35–48. ACM Digital Library, 2005.

[38] H. Susan. Precise flow-insensitive may-alias analysis is np-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, Jan. 1997. ISSN 0164-0925. doi: 10.1145/239912.239913. URL http://doi.acm.org/10.1145/239912.239913.

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

[40] S. Tobin-Hochstadt and V. St-Amour. The typed Racket guide. http://docs.racket-lang.org/ts-guide/.

[41] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, Dec. 1987. doi: 10.1145/38765.38828.

[42] T. Wang and S. F. Smith. Precise constraint-based type inference for Java. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Proceedings ECOOP '01*, volume 2072 of *LNCS*, pages 99–118, Budapest, Hungary, June 2001. Springer-Verlag.