Onward! 2014 – October 20–24, 2014, Portland, OR, USA

Mining the Ecosystem to Improve Type Inference For Dynamically Typed Languages

Boris Spasojević, Mircea Lungu, Oscar Nierstrasz

Software Composition Group
University of Bern
Switzerland
{spasojev,lungu,oscar}@iam.unibe.ch

Abstract

Dynamically typed languages lack information about the types of variables in the source code. Developers care about this information as it supports program comprehension. Basic type inference techniques are helpful, but may yield many false positives or negatives.

We propose to mine information from the software ecosystem on how frequently given types are inferred unambiguously to improve the quality of type inference for a single system.

This paper presents an approach to augment existing type inference techniques by supplementing the information available in the source code of a project with data from other projects written in the same language. For all available projects, we track how often messages are sent to instance variables throughout the source code. Predictions for the type of a variable are made based on the messages sent to it.

The evaluation of a proof-of-concept prototype shows that this approach works well for types that are sufficiently popular, like those from the standard librarie, and tends to create false positives for unpopular or domain specific types. The false positives are, in most cases, fairly easily identifiable. Also, the evaluation data shows a substantial increase in the number of correctly inferred types when compared to the non-augmented type inference.

Categories and Subject Descriptors D.2.3 [*Coding Tools and Techniques*]: Object-oriented programing; D.3.m [*Miscellaneous*]

Keywords Ecosystem Mining; Type Inference

1. Introduction

Software developers spend more time on maintaining and evolving existing software than writing new code. Maintenance consumes over 70 percent of the total life-cycle cost of a software product [4]. This means that support for reading and understanding code is very important. Static type information in source code helps developers understand how the software system works [15], but the expressiveness provided by dynamically typed languages can make developers more productive [10]. Many attempts have been made at getting the best of both through type inference or optional typing.

Most type inference techniques rely on statically analysing the source code of the software system in question, and using the gathered data to infer the possible types. A basic approach is to track the usage of a variable *i.e.*, the messages sent to it, and infer the type by determining which classes implement the corresponding methods. This can lead to false positives, *i.e.*, types that match the required interface but can never actually be reached at run time. More advanced techniques perform deeper analysis *i.e.*, using data flow or control flow, but ultimately suffer from the same problem of false positives. A developer faced with a provided set of possible types cannot easily identify the false positives.

This paper presents an approach to augment existing type inference techniques by supplementing the information available in the source code of a project with data from other projects written in the same language. Since programs written in the same language often share dependencies we will consider as the background of this work a corpus of systems that belong to the same software ecosystem [13]. By using the data from the ecosystem, it is possible to increase the amount of information used to infer types and thus help avoid and identify potential false positives. For all available projects from the ecosystem, we track how many times messages are sent to instances of available types throughout the source code. With this information, we can sort the potential types of a variable the developer cares about based on their likelihood of being the actual type in the context. The likelihood 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.

We have implemented a proof-of-concept prototype and used it to evaluate the approach. We show that, for our implementation, measuring the frequency of association between a message and a type throughout the ecosystem source code is helpful in identifying correct types. The evaluation data shows a substantial increase in the number of correctly inferred types when compared to the type inference working only on data from one project.

The paper is organized as follows: section 2 gives a high level overview of the problem and the proposed approach to solving it; section 3 presents the related work of type inference and mining software repositories; section 4 gives a detailed description of the proposed approach, as well as the formal model used to describe it; section 5 presents the details of the implementation of the prototype; section 6 shows the methods and results of the evaluation of the prototype; and finally section 7 concludes and discusses future work.

2. Overview

To better understand the contributions of this paper, we take a look at an existing three-step approach to type inference [20]. We start from this approach because it is simple to understand and implement, is reasonably fast and is representative of its field. Other more complex approaches would gather more data about the system to increase precision, and such complexity is unneeded for the purpose of this paper.

The approach has three steps:

1. Interface type extraction. This phase reconstructs the type of a variable of interest by using static analysis to find all messages sent to it within the context of the given class. The system is then searched for all classes that implement this set of messages.
2. Assignment type extraction. This phase reconstructs the type with respect to the assignments to the variable. This is a heuristic based analysis of the right side of assignments to the variable in question.
3. Merger. Merging the results from phases one and two into the final type results for the variable. Several different ways exist to do the merge [20], but we focus on the one that gives priority to the assignment type, and moves to interface types if an assignment type does not exist .

This "single-system type inference" (SSTI), is not helpful in cases where the amount of data gathered by the first two phases is limited. To understand this limitation consider the example in Listing 1.

The example[1] is written in Pharo Smalltalk[2]. The first part (lines 1–7) declares a new class called MethodBrowser and lists the instance variables of the class. Special attention for this example is put on the instance variable toolbarModel. The second part is the definition of a method named initializePresenter. This method is the only place toolbarModel is used. The only usage is sending it the message method:.

```
1  ComposableModel subclass: #MethodBrowser
2    instanceVariableNames:
3                'listModel
4                textModel
5                toolbarModel'
6    category:
7                'Spec−Examples−PolyWidgets'
8
9  MethodBrowser>>initializePresenter
10   listModel whenSelectedItemChanged: [:selection |
11      selection
12
13         ifNil: [
14            textModel text: ''.
15            textModel behavior: nil.
16            toolbarModel method: nil ]
17
18         ifNotNil: [:m |
19            textModel text: m sourceCode.
20            textModel behavior: m methodClass.
21            toolbarModel method: m ]].
22
23   self acceptBlock: [:t |
24               self listModel selectedItem inspect ].
25   self wrapWith: [:item |
26               item methodClass name,'>>#', item
      selector ].
```

Listing 1. The type of toolbarModel cannot be detected by the single-system approach

Suppose the developer needs to know the type of the toolbarModel instance variable. She could care about this in order to understand which of the implementations of method : will be invoked when this code is executed or just use the knowledge of the type of this variable to better understand the entire system. In Smalltalk, good practices recommend instance variable names to match the type of the variable. However, we find no ToolbarModel class in the system.

Applying the previously described approach to this instance variable would produce 21 possible classes. This is due to the fact that there are no assignments to the variable and only one method invocation, and the method in question is defined in all 21 classes. This means that if the developer wishes to understand which implementation of method: is invoked, she is in an uncomfortable position of having 21 possibilities. The actual number of possible classes is even larger, because we ignore all subclasses of the classes defin-

[1] The code snippet is actual code from the Spec system, to be found at `http://smalltalkhub.com/#!/~Pharo/Spec`

[2] `http://www.pharo-project.org`

ing the method in question. Obviously in cases like this, the information provided to the developer is not helpful. Pluquet *et al.* show that on average less than 40% of instance variables from their evaluation receive enough messages and initializations to successfully infer one possible type [20].

Given a set of possible types provided by the SSTI it would be helpful to the developer if the set were sorted by how likely each of the types is to be correct. Since the data we have available is a set of selectors[3] that the instance variable receives, we can compare this set to patterns of message sending in other projects from the ecosystem. The intuition is that the more often we find that the same messages are sent to a uniquely identifiable type, the more likely that type is to be correct.

In our case, after the analysis of other systems we find out that the message method: is commonly sent to instances of 3 classes out of the 21 that implement the method. Those are MethodToolbar, ZnRequest, and SourceMethodConverter. The actual type assigned to the instance variable toolbarModel at run time is MethodToolbar. In this paper we argue that type association information from other projects can be beneficial to recovering types.

The proposed approach (Ecosystem-aware type inference — EATI) automates the process of using ecosystem data for augmenting SSTI and it consists of two phases. The first is an analysis of a large number of systems that results in the data about the frequency of association of messages and types. The second phase concerns the developer in need of type inference. To infer a type, we attempt SSTI and in case the results are ambiguous we query the data from phase one to receive a collection of possible types sorted by the number of times the messages were sent to types in the ecosystem.

3. Related work

Two field of related work are relevant to EATI, namely type inference and large scale software analysis.

3.1 Type Inference

EATI addresses *type inference for dynamically typed object-oriented languages* to support program comprehension. The scope of this work is different from classical type inference techniques for statically typed languages like Scala [18], where type inference frees the developer from having to specify types that can be inferred.

Much of modern type system research is based on the work of Milner who published the description of a polymorphic type-inference algorithm called "Algorithm W"[16]. It is a fast algorithm, performing type inference in almost linear time with respect to the size of the source code and was first implemented as part of the type system of the programming language ML.

[3] In Smalltalk jargon, a *selector* is the name of a message, *i.e.*,+ or method:, used to select the *method* to respond to the message.

A well known type inference algorithm is the Cartesian Product Algorithm [1] (CPA). This algorithm infers concrete types to support performance of parametric polymorphism. CPA does this by partitioning the calling context of a method based on the types of the actual arguments passed to the method. It supports dynamically typed languages, as it is implemented originally for Self, and its contribution is limited to inferring concrete types from polymorphic types. CPA is the basis for other type inference engines for other languages *i.e.*, Starkiller [25], a type inferencer and compiler for Python.

A fast type inference technique presented by Pluquet [20] is used as a basis for the prototype implementation of the type inference presented in this paper. This technique is outlined in the motivating example section.

The approaches so far use different kinds and quantities of data obtained through statical analysis to infer types. None of them expand to more then one system, so any of them can benefit from EATI. An implementation of EATI on top of these approaches would need to be significantly different from the one presented in this paper, as it would have to manage the different data used.

Other approaches use the execution of a program to gather types. One such approach is presented by Jong-hoon *et al.* for the language Ruby [6]. This approach uses wrappers for variables that generate constraints during execution which are later used to infer types. The inferred types can be used for documentation, and thus better code comprehension, but all dynamic approaches are limited by the requirement that the code needs to be runnable, either through test cases or symbolic execution [8].

Another field of related work has to do with optional typing. Optional typing attempts to enable developers all the benefits of using dynamically typed languages, with the option of specifying types when and where they deem appropriate [2]. This enables compile-time type checking for provided types, and also enriches the source code with static types. Examples of optionally typed languages are Strongtalk and Gradualtalk — dialects of smalltalk, Dart — a language developed by Google, and Typescript — an optionally typed Javascript developed by Microsoft. A type inference engine for these languages could be used to infer and generate the optional types. This would free the developer from specifying inferable types, as with Scala, and such engines could also benefit from ecosystem data.

3.2 Large Scale Software Analysis

A large body of work is concerned with mining open source software repositories to address a variety of software development issues: from predicting which parts of the system are likely to have defects [5], to automatically detecting code clones across open source systems [24], supporting code search across the web or large collections of software projects [3] and automatically detecting the license of jar archives [7].

One particular direction of research related to ours is the work on API specification mining [23]. In order to detect groups of methods that are usually called together Nguyen *et al.* [17] statically analyse method call and field access graphs. The mined API patterns represent sets of methods that are called on a single object, thus making the part which extracts the protocol similar to ours. Pradel and his colleagues used a combination of static and dynamic analysis to automatically detect illegal uses of APIs [21] while building multi-object protocols.

To support developers writing code with APIs they are not familiar with, the Strathcona tool automatically searches a given corpus of systems and finds relevant contextual source code examples [11]. The developer indicates a relevant code fragment as input for the tool which then fetches and displays a series of relevant code examples which the developer can browse.

Ossher *et al.* present a method for automatically resolving dependencies for open source software which works by cross-referencing the missing type information in a project with a repository of candidate artifacts [19]. They — similar to us — build a detailed AST for every individual system they analyze. The AST is used to analyze an individual project and detect the missing types. Once the missing types have been identified, the final step is to match them against the artifacts in the candidate repository. In our previous work we have also analyzed inter-system dependencies by analyzing the method call graph, identifying the targets of calls which do not exist inside of a system, and detecting these targets elsewhere in the ecossytem [14].

In their work, Thummalapenta and Xie crawl the web for methods and classes that are often reused [26]. They collect the frequency of calls to methods and classes of a given API in order to recognize so-called hotspots. The information that they collect is similar to ours; the difference is in the end-goals — they support the understanding of an API while we aim at improving the type inference. Unlike their, and most of the related work which is targetted at Java systems, we have to address challenges inherrent to dynamically typed programming languages.

Robbes *et al.* analyzed an entire Smalltalk ecosystem to understand the way in which changes in one system propagate to the systems downstream [22]. In their case they analyze the entire ecosystem and stored the information in the *Ecco* meta-model. They discard the information regarding the object that receives a certain message, thus that meta-model is simpler than the one we use in this work. To build the model they analyze method calls but their analysis is limited to building the AST of a single method for data extraction. This results in a less rich model but enables them to analyze any kind of project, even the ones that do not compile. In their work they analyze all the available versions of all the systems in the ecosystem; in our work we do not make use of evolutionary information.

4. Ecosystem-aware Type Inference

To explain EATI we introduce a simple set-theoretic model in Figure 1 that captures key properties for the entities shown in the UML diagram in Figure 2. For simplicity we ignore temporary variables and method arguments throughout the paper. We can greatly simplify the model and implementation by ignoring them, and since the approach works the same for these variables application of the approach to them is trivial.

4.1 Core model

Given all the source code in a software ecosystem, C is the set of all classes, F the set of all instance variables (*i.e.*, fields), M the set of all methods, and S the set of all "selectors" (*i.e.*, method names).

$$\mathit{def}_f : F \to C \tag{1}$$

$$\mathit{def}_m : M \to C \tag{2}$$

$$\mathit{sup} : C \to C \tag{3}$$

$$\mathit{sel} : M \to S \tag{4}$$

$$\mathit{sends} : M \times F \to 2^S \tag{5}$$

$$\text{where } \mathit{def}_f(f) \notin \mathit{sup}^*(\mathit{def}_m(m)) \Rightarrow \mathit{sends}(m, f) = \emptyset \tag{6}$$

Figure 1. The core model. F = fields, C = classes, M = methods, S = selectors.

A given field f is uniquely defined in a class $c = \mathit{def}_f(f)$ (Equation 1). Similarly, each method m is defined in a unique class $c = \mathit{def}_m(m)$ (Equation 2). Every class c other than Object has a unique superclass $c' = \mathit{sup}(c)$ (Equation 3). *sup* is a partial function, since $\mathit{sup}(\mathsf{Object}) = \bot$.

Every method m has a unique method name (*i.e.*, a *selector* used select m) $s = \mathit{sel}(m)$ (Equation 4), and every method m sends a set of messages selectors to a given field f, namely $\mathit{sends}(m, f)$ (Equation 4).

Note that a method m can only access fields defined in the same class where m is defined, or fields inherited from one of its superclasses. For all other combinations of m and f, $\mathit{sends}(m, f)$ returns the empty set (Equation 6).

Consider the example class hierarchy Figure 3. In the example, $\mathit{def}_f(\mathsf{rect}) = \mathit{def}_f(\mathsf{tri}) = \mathit{def}_m(\mathsf{main}) = \mathsf{DrawEditor}$, as both these instance variables and the method main are defined in the class DrawEditor. Also, $\mathit{sup}(\mathsf{DrawEditor}) = \mathit{sup}(\mathsf{Shape}) = \mathsf{Object}$ which is obvious from the hierarchy.

We can now query the model to compute metrics that will allow us to rank the results of type inference, as summarized in Figure 4. The *interface* of a class c, $\mathit{ifc}(c)$, is the set[4] of selectors of all methods defined in c and its superclasses

[4] Note that we implicitly extend *sel*, def_m and other functions in the usual way to take sets of values as arguments and similarly return sets of values.

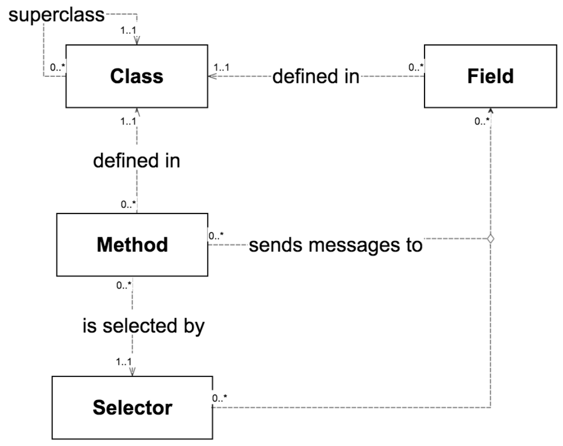

Figure 2. The core model in UML.

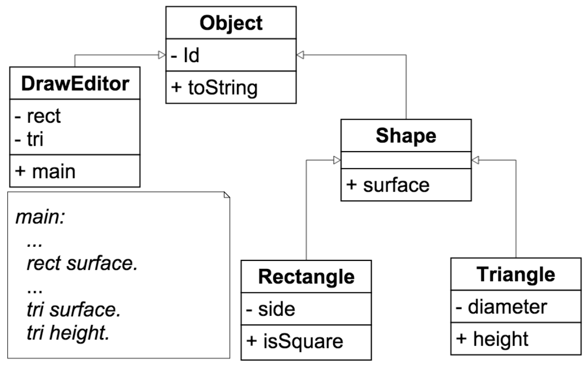

Figure 3. Sample class hierarchy with the implementation of one method .

Equation 7. The selectors *received* by a field f, $\mathit{rec}(f)$ is the set of all selectors of messages sent to f by methods defined in the same class $c = \mathit{def}_f(f)$ (Equation 8)[5].

Returning to the example, the interface of the class Rectangle is:

$$\mathit{ifc}(\mathsf{Rectangle}) = \{\mathsf{isSquare}, \mathsf{surface}, \mathsf{toString}\}$$

as those are all the methods defined in it and its superclasses. Looking at the implementation of method main we can see that messages are sent to the instance variables rect and tri. We can express which messages with

$$\mathit{rec}(\mathsf{rect}) = \{\mathsf{surface}\}$$

and

$$\mathit{rec}(\mathsf{tri}) = \{\mathsf{surface}, \mathsf{height}\}$$

[5] This definition is consistent with classical SSTI [20]. Later we will consider messages sent by methods in subclasses as well.

$$\mathit{ifc}(c) \equiv \mathit{sel}(\mathit{def}_m^{-1}(\mathit{sup}^*(c))) \tag{7}$$

$$\mathit{rec}(f) \equiv \mathit{sends}(\mathit{def}_m^{-1}(\mathit{def}_f(f)), f) \tag{8}$$

$$\mathit{types}(f) \equiv \{c \in C | \mathit{rec}(f) \subseteq \mathit{ifc}(c)\} \tag{9}$$

$$\mathit{roots}(C') \equiv \{c \in C | \forall n > 0, \mathit{sup}^n(c) \notin C'\} \tag{10}$$

$$\mathit{unique}(f, c) = \begin{cases} 1 & \mathit{roots}(\mathit{types}(f)) = \{c\} \\ 0 & o/w \end{cases} \tag{11}$$

$$\mathit{selscore}(c, s) = \sum_{f \in F, s \in rec(f)} \mathit{unique}(f, c) \tag{12}$$

$$\mathit{classscore}(c, f) = \sum_{s \in rec(f)} \mathit{selscore}(c, s) \tag{13}$$

Figure 4. Computing class scores over the core model.

The set of possible *types* of a field f, $\mathit{types}(f)$, is the set of classes whose interface includes all selectors received by f (Equation 9). The roots of the hierarchies of a set of classes C' is the subset of those classes without superclasses in C' (Equation 10).

Applying this to our example we get

$$\mathit{types}(\mathsf{rect}) = \{\mathsf{Shape}, \mathsf{Rectangle}, \mathsf{Triangle}\}$$

and

$$\mathit{types}(\mathsf{tri}) = \{\mathsf{Triangle}\}$$

A field f is inferred to be of a *unique* type c if set of inferred types of f has a unique root c. The function $\mathit{unique}(f, c)$ returns a count of 1 for all such fields (Equation 11). This means that $\mathit{unique}(f, c)$ will, in our example, equal 1 in only two cases.

$$\mathit{unique}(\mathsf{rect}, \mathsf{Shape}) = 1$$

$$\mathit{unique}(\mathsf{tri}, \mathsf{Triangle}) = 1$$

Now we compute the *selector score* of a class c with respect to a selector s, $\mathit{selscore}(c, s)$, as the number of fields f that are determined to be of the unique type c, where s is sent to f (Equation 12). A few selector score values for combinations of classes and selectors from our example are

$$\mathit{selscore}(\mathsf{Shape}, \mathsf{surface}) = 1$$

$$\mathit{selscore}(\mathsf{Rectangle}, \mathsf{surface}) = 0$$

$$\mathit{selscore}(\mathsf{Triangle}, \mathsf{height}) = 1$$

Finally, we compute the *class score* of a given class c with respect to a field f, $\mathit{classscore}(c, f)$, as the sum of all its selector scores for selectors of messages sent to f

(Equation 13). The usage of class score is beyond the scope of our small example, but will be explained further in the paper.

4.2 Storing data from the ecosystem

The first step to the proposed approach is to gather type information from projects in the ecosystem and information on message sending to instances of those types. We store the gathered data in a central repository, so it can be queried when necessary.

An entry in the stored data consists of a class name $c \in C$, selectors $\{s_1...s_n\} \subseteq S$ sent to instances of c, and the number of times each $s_i \in \{s_1...s_n\}$ has been sent to instances c. This number is called the selector score of a class ($selscore(c, s)|s \in S, c \in C$), as messages with the same selector can be sent to instances of different classes, yielding different selector scores. A sample database is presented in Table 1. In this paper we show that this information is sufficient for improving the type inference.

The data needed for EATI can be gathered through dynamic or static analysis. In this context, dynamic analysis means gathering type information from a running system [6]. This provides actual run time types, but requires the system to be runnable and produces false negatives — a variable may not actually be bound to a type during the observed execution of the system, but might during others. Static analysis means running a type inference engine on the source code. As we saw in the example from the beginning of section 2, static analysis can often produce false positives — types deemed "possible" that never occur at run time. Since the ecosystem data is large, we can ignore the false positives and false negatives and store the remaining data.

4.3 Using the stored data

In order to infer types, we apply SSTI, and in case there is more than one possible type, the data gathered from the ecosystem is queried for more information in order to sort the possible types and present the developer with the more likely candidates. The repository query for an instance variable $f \in F'$ should contain $rec(f)$, where F' is the set of all instance variables in the system being typed by the user. The result of the query is a set of possible classes $\{c_1...c_n\} \subseteq C$, determined by which classes in the repository have records of their instances receiving selectors from $rec(f)$. The result of the query should be sorted by the class score.

For example, given the repository table from Table 1, querying the repository with a set of selectors {substring:, startsWith:} would return just the class ByteString with a score of 23. ByteString is the only result because it is the only class found in the repository whose instances received the given selectors. The score is due to the selector substring: having a score of 9 and the selector startsWith: having a score of 14, yielding 23.

If the query contained only the selector $+$, the result would have been a set of classes containing the class Integer with a score of 27 and the class ByteString with a score of 10. Instances of both classes have been observed to receive this selector, but instances of class Integer receive it more often.

Class Name	Selector	Score
ByteString	substring:	9
	toUpperCase	6
	+	10
	startsWith:	14
	endsWith:	4
Integer	+	27
	−	63
	toString	40
	bitAnd:	2

Table 1. A sample repository containing information on the frequency of sending certain selectors to classes ByteString and Integer

5. Implementation

We have implemented a prototype of EATI for Pharo Smalltalk. We chose Smalltalk because it is highly reflective, and enables fast and easy development of analysis tools [9]. The implementation consists of three distinct subsystems: a data gatherer that mines usage data from the ecosystem, a GUI available to the developer, and a persistent store through which the other two subsystems communicate.

5.1 Data gathering

The data gatherer is tasked with populating a database with information on type-selector relationships from the ecosystem. As an input, it requires access to repositories of a number of software projects. After loading the source code of a project, the gatherer proceeds to run a type inference engine on the entire project source code.

The type inference engine used is an improved implementation of the SSTI described in the example from the beginning of section 2 that takes into account field usage data from the subclasses. This provides more information for instance variables that are used predominantly in the subclasses of the class that declares it and requires a small modification to our model redefining rec as

$$rec(f) \equiv sends(M, f)$$

for all $f \in F$. Note that this definition will give empty results for all m where f is not accessible, so only methods of the class $c = def_f(f)$ and its subclasses actually contribute selectors to the result.

The prototype iteratively pre-computes $selscore(c, s)$ for all classes and selectors, starting with the class Object, the root of the class hierarchy. Perfroming a depth-first search of the class hierarchy ensures that all the data from the subclasses of each class are gathered. Once the entire subtree for a particular class has been traversed, the search for adequate types of the instance variables of the class begins, based on the data collected from the subtree.

The gatherer runs on a timed schedule to ensure that the data in the store is up-to-date. The gatherer is run regularly and all the data in the store is rebuilt. Newer versions of the projects from the ecosystem should contain more data, and the new store should be superior to the old one. This also ensures that the entire system is aware of changes in the ecosystem, such as the introduction of new libraries.

5.2 The store

The pre-computed *selscore* are represented as a set of triplets:

$$(c, s, \mathit{selscore}(c, s)) | c \in C, s \in S$$

and stored as a key-value JSON-document. We group all the triplets with the same c, and use c as the key for that entry in the database. A textual representation of a sample JSON-document follows.

```
{
  "_id" : ObjectId("51b0df6b44b1392c8e7ee0ec"),
  "className" : "Mutex",
  "selectors" :
  {
    "critical:" : 2,
    "ifNil:" : 1
  }
}
```

By the end of the analysis of all the projects the database contains the global score for every available class-selector combination. We use a MongoDB[6] database to store the data.

5.3 The client

The client is the front end of the entire system, and is the bridge between the user and the system. We implemented a very simple GUI that offers the developer the choice which class should be processed, and whether or not to include its subclasses in the analysis. Processing the class consists of running SSTI on its instance variables, consulting the database when necessary and presenting the results to the developer. There are many ways to use the provided data both by tools and by the developer, but the client side usages are beyond the scope of this paper.

6. Evaluation

To evaluate the prototype implementation we populate the store with data from 74 open source projects from the Pharo Smalltalk ecosystem. The projects were chosen based on their availability and ease of automated access. Automated access is important because their source code needs to be loaded and analysed by a tool during the data gathering phase. The gatherer loads the source code of all the projects using the configuration browser. The configuration browser is a tool to automatically load Smalltalk project source code and dependencies, similar to Maven[7] for Java.

A total of 8374 classes were analyzed. This produced 746 entries in the store. This means that running type inference on all the instance variables of the classes produced 746 classes $\{c_1...c_{746}\} \subset C$ such that

$$\exists f \in F | \mathit{unique}(f, c) = 1, rec(f) \neq \emptyset$$

After populating the store with data gathered from the projects, we take 97 instance variables from 5 projects. These projects are not a part of the set of used to populate the store and were chosen because they have unit tests available. We where limited to these 5 projects because of the relatively small size of the Pharo ecosystem and availability of projects with unit test coverage is small.

The run-time types of the instance variables are recovered by instrumenting the source code of the projects to log types of objects assigned to instance variables and running the unit tests. These types are held to be the actual types, implications of which are described in the threats to validity section.

Types of these instance variables are then inferred using SSTI and EATI.

It should be noted that the projects used for testing contain a total of 402 instance variables, but most of them were ignored for one of two reasons:

1. A total of 107 instance variables received no messages thus the instance variable is not a valid candidate for this type inference technique;
2. Running the unit tests did not provide a run-time type for 198 instance variables. This is most likely due to poor test coverage.

Throughout the evaluation we try to answer the following questions: How well does SSTI work, what is the improvement with EATI, and when and why does EATI fail? The evaluation discussion is divided into 3 parts, one focusing on discussing successfully inferred types, one focusing on the false positives and the last commenting on the remaining situations in which the single-system approach failed and no data was provided by the ecosystem-aware approach. A summary of the evaluation results is given in Table 2.

6.1 Successful attempts

EATI provides a sorted list of the most likely types of an instance variable. In the best case scenario the correct type

[6] http://www.mongodb.org

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

Total	Successful		False positives		No data
	Single system	Eco-aware	Selectors only from *Object*	True failures	
97	21	20	23	25	8

Table 2. Summary of the evaluation results

should be at the top of the list. We declare two scenarios for a successful attempt:

1. if the SSTI infers the correct type (as provided by running the unit tests)
2. if the SSTI provides several types and the correct type is at the top of the sorted list provided by EATI.

The results show 21 instance variable types that have been successfully inferred using SSTI and an additional 20 using EATI. This means that using the EATI almost doubled the number of successfully inferred types.

An analysis of the inferred types shows that EATI works reliably for types from the standard library, as 19 of the 20 instance variables have run-time types from the standard library. These include *Array*, *SmallInteger*, *ByteString* and others. We argue the approach works well with the standard library because it is so widely used, and the data on type-selector relations is abundant. We expect that the success would generalize to any types sufficiently popular in the ecosystem.

6.2 False positives

For an inference to be considered false positive the types provided by the EATI should be different than the actual run-time type of the instance variable (as provided by running the unit tests). This situation can be more damaging to the comprehension of the source code than not receiving any result at all. This is because it may lead the developer to make wrong decisions based on the wrong type of a variable.

The results show 48 false positives. The number seems unacceptably high, but a second look at the data reveals that almost half of the false positives were caused by the lack of selectors sent to those instance variables. A total of 23 instance variables that caused a false positive received only selectors declared in the Object class, such as the ifNil : selector used to check if the object in question is a nil object[8]. Selectors declared in the Object class, can be sent to any Smalltalk object. Thus, those selectors carry no useful type information. We argue that those false positives can be ignored, as they would very easily be identified by checking if all the given selectors are defined in Object, which can be easily automated.

[8] nil is an object in Smalltalk. It is the sole instance of the UndefinedObject class.

Out of the remaining 25 false positives, 6 fail to give the correct type at the top of the list, but the correct type is present in the top three. We call these “near misses”.

The remaining 19 false positives each fall into one of two categories:

1. Run-time types not present in the ecosystem — These types are specific to the project in question, and as such are not present in the store *i.e.*, classes only used within this project. Since EATI can not access source code that uses these classes, they can only be identified through SSTI.
2. Domain specific selectors and types — this false positive arises when selectors used widely for one purpose are used in a different or domain specific manner. For example, the comma selector (“,”) when sent to an object of type ByteString is used to concatenate strings. On the other hand, in the PetitParser [12] framework this selector is used to create a sequence of parser combinators. Since concatenation of strings is far more frequent then parser combinator sequences, the data in the store suggests that the type of a variable receiving only the selector , is a ByteString. At this point, it is left to the developer to use her knowledge of the specifics of the project in question to detect this kind of false positive. In future work we would like to explore whether additional context information can be exploited to determine the domain of the project, and adjust the EATI accordingly.

6.3 No data from the EATI

A total of 8 instance variables were completely unidentifiable. The selectors sent to these variables are declared in more then one class in the system, thus SSTI results in many false positives. The combination of these selectors has never been linked to a type in the store, *i.e.*,:

$$\nexists c \in C | \mathit{unique}(\mathsf{f}, \mathsf{c}) = 1$$

This does not mean that all individual selectors from *rec*(f) have never been seen in the ecosystem. It means that during the data gathering phase no uniquely inferred type has received this combination of selectors.

These situations leave the developer with no insight into the type of a variable, but are also not damaging as they do not mislead like the false positives do.

6.4 Threats to validity

Even though EATI is applicable to any dynamically typed language we cannot guarantee the evaluation will generalize to other ecosystems. Although the approach should benefit type inference in any ecosystem, we cannot state in what way without more insight.

The main threats to validity of our evaluation come from the projects that were used. One threat is that the 74 selected projects are not truly representation of the ecosystem. In the selection process we attempted to address this threat by identifying projects covering a wide range of domains and sizes.

Another threat is the fact that unit tests are used to determine the run-time types for instance variables. Unit tests do not provide a complete picture of the running system, and their execution provides a limited set of possible types for instance variables. With that in mind, the effort to precision ratio for using unit tests is high, and the alternatives of symbolic execution or manually running the systems are significantly more difficult, and are also prone to false negatives. It is an open question to determine whether a given set of unit tests offers a representative picture of the actual types that would be bound during typical (non-test) runs.

It is possible that the selection of SSTI we made could have affected the results. Even though the SSTI we used is representative it is hard to say what effect using other techniques to enable EATI would have on the results. We chose this SSTI for its simplicity and speed, and as such it served the needs of this evaluation. Since EATI is meant as a supplement for SSTI, changing the SSTI engine could increase the SSTI success rate and make EATI seem less potent. On the other hand, if another SSTI were used in the data gathering phase of EATI, the results might normalize. In future work we plan to explore the impact of EATI on other SSTI approaches.

Throughout the paper we ignore how the fact that different versions of APIs coexisting in the ecosystem could affect the results. The magnitude and implications of this escape the scope of the paper.

Finally, using different type inference engines in the data gathering phase and on the client side could yield very different result.

7. Conclusion and future work

This paper presents a novel approach to type inference that supplements the information available in the source code of a project with data from other projects written in the same language. The approach requires a large set of projects from the same ecosystem to be analyzed, statically or dynamically and indexes the times selectors are associated to different types. After a gathering phase, we store the findings, and a client that infers more than one possible type for a variable can consult the data to sort the candidates and identify the most likely ones.

The prototype implementation, written for the Pharo Smalltalk ecosystem, enables an analysis of the pros and cons of the approach. The approach has shown to be particularly useful in inferring standard types (SmallInteger, Boolean, ByteString, *etc.*), and the collection types (Set, Dictionary, *etc.*). We conclude that this is due to the massive usage of these types throughout the ecosystem, and hypothesize that the approach applies to any sufficiently popular type.

In the situations where the approach fails, three patterns can be identified.

Firstly, the situations where no relevant selectors are sent to the instance variable. In these cases the approach will offer either no solution or an unhelpful one (*i.e.*, Object class). We conclude that these cases are easily identifiable by presenting to the developer not just the type recommendations but also the set of selectors sent to the instance variables in question. If the selectors are too few or two generic (*i.e.*, the ones defined in the Object class) the developer or a tool will be able to conclude that the recommendations are based on loose data.

Secondly, the approach does not work on types that are specific to the project in question. For all the types used only in the project, the ecosystem data is useless as it is completely oblivious to the existence of these types. If these types cannot be inferred by the data available in the project that defines them, they receive no benefit from ecosystem data either. Such cases can be easily recognized.

Finally, in situations where selectors are commonly used for operations on one type, but less commonly for other operations on different types, the approach will favor the more frequent one in all situations. We conclude that these cases are not faulty, as the EATI generates recommendations solely based on the highest count of messages sent to types. Once again, it is up to the developer to be aware of the domain of the project and the issues that may arise from it. As future work, we plan to explore how knowledge of the project domain can be used to automatically adjust the rankings.

Throughout the analysis of the approach described in this paper, several opportunities for improvement have arisen. With more engineering effort it would be possible to solve some of the shortcomings, *i.e.*, allowing the developer to specify the domain or by attempting to infer the domain automatically.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Agile Software Assessment” (SNSF project No. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015). We also gratefully acknowledge the financial support of the Swiss Group for Object-Oriented Systems and Environments (CHOOSE) 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] E. Allende, O. Callau, J. Fabry, É. Tanter, and M. Denker. Gradual Typing for Smalltalk. *Science of Computer Programming*, Aug. 2013. . URL `http://hal.inria.fr/hal-00862815`.

[3] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Sci. Comput. Program.*, 79:241–259, Jan. 2014. ISSN 0167-6423. . URL `http://dx.doi.org/10.1016/j.scico.2012.04.008`.

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

[5] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, pages 1–47, 2011. ISSN 1382-3256. . URL `10.1007/s10664-011-9173-9`.

[6] 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. .

[7] M. Di Penta, D. German, and G. Antoniol. Identifying licensing of jar archives using a code-search approach. In *Mining Software Repositories (MSR), 2010 7th IEEE WorkingConference on*, pages 151–160, May 2010. .

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

[9] 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.

[10] 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. . URL `http://doi.acm.org/10.1145/1932682.1869462`.

[11] R. Holmes. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Softw. Eng.*, 32(12):952–970, 2006. ISSN 0098-5589. . URL `http://dx.doi.org/10.1109/TSE.2006.117`.

[12] J. Kurs, G. Larcheveque, L. Renggli, A. Bergel, D. Cassou, S. Ducasse, and J. Laval. PetitParser: Building modular parsers. In *Deep Into Pharo*, page 36. Square Bracket Associates, Sept. 2013. ISBN 978-3-9523341-6-4. URL `http://www.deepintopharo.com/`.

[13] M. Lungu. *Reverse Engineering Software Ecosystems*. PhD thesis, University of Lugano, Nov. 2009. URL `http://scg.unibe.ch/archive/papers/Lung09b.pdf`.

[14] M. Lungu, R. Robbes, and M. Lanza. Recovering inter-project dependencies in software ecosystems. In *ASE'10: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. ACM Press, 2010. . URL `http://scg.unibe.ch/archive/papers/Lung10a.pdf`.

[15] 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. . URL `http://doi.acm.org/10.1145/2398857.2384666`.

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

[17] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-001-2. . URL `http://doi.acm.org/10.1145/1595696.1595767`.

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

[19] J. Ossher, S. Bajracharya, and C. Lopes. Automated dependency resolution for open source software. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 130 –140, may 2010. . URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5463346`.

[20] 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. .

[21] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. Statically checking api protocol conformance with mined multi-object specifications. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 925–935, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL `http://dl.acm.org/citation.cfm?id=2337223.2337332`.

[22] R. Robbes, M. Lungu, and D. Roethlisberger. How do developers react to API deprecation? The case of a Smalltalk ecosystem. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE'12)*, pages 56:1 – 56:11, 2012. . URL `http://scg.unibe.ch/archive/papers/Rob12aAPIDeprecations.pdf`.

[23] M. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated api property inference techniques. *Software Engineering, IEEE Transactions on*, 39(5):613–637, 2013. ISSN 0098-5589. .

[24] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74:470–495, May 2009. ISSN 0167-6423.

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

[26] S. Thummalapenta and T. Xie. Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 327–336, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-2187-9. . URL `http://dx.doi.org/10.1109/ASE.2008.43`.