# VM Support for Live Typing

Automatic Type Annotation for Dynamically Typed Languages

Hernán Wilkinson
Software Development
10Pines SRL, Argentina
hernan.wilkinson@10pines.com

## ABSTRACT

Live Typing automatically annotates variable types based on the objects assigned to them, and method return types based on the returned objects. IDEs can exploit this information to greatly improve the development tools and the programming experience, bringing some of the benefits of Static Typing tools to Dynamically Typed Languages.

Live Typing benefits are boosted in Live Development Environments because the same VM is used to run the system under development and the IDE. Type information can be obtained from the running code no matter if that code is part of the language's core, the development tools, or the system under development. The more code is run the more type info it will provide.

For Live Typing to be usable, the VM has to collect the information as fast as possible, in objects that have to be accessed and configured from the development environment. This paper presents how the OpenSmalltalk-VM was modified to accomplish this task and the challenges we have for future versions.

## CCS CONCEPTS

• Software and its engineering → Virtual machines; Object oriented languages; Development frameworks and environments

## KEYWORDS

Virtual Machine, Live Typing, Static Typing, Dynamic Typing, Live Development Environments

## 1 INTRODUCTION

Live Typing [1] is a technique that uses the fact that Live Development Environments as Cuis [2], Pharo [3], Squeak [4], and all Smalltalk-80 [5] related languages, run on the same VM that the programmer uses to test and run the system under development. That trait differs from other VM-based programming languages like Ruby, Python and Java among others, where the development environment runs on a VM while the execution of the system under development and its tests run on another VM.

Almost all Meta-Circular Languages [6,7,8] have Live Development Environments with a VM that can share runtime information with the development tools. Among that information is type information, which it is not limited to the system being developed but it also covers all the running code in the VM. That includes the language's compiler, core libraries as well as development tools, such as the debugger or code editor.

Live Typing is currently implemented in the stack version of OpenSmalltalk-VM [9] and in the Cuis Smalltalk dialect. Bytecodes that store objects to variables and returns from execution contexts were modified to keep the classes of the assigned/returned objects in an array of types that can be queried and modified from the development environment.

The implementation was done with the following requirements in mind:

- Most of the work should be done in the development environment. The least the VM does, the better.
- The development environment should be usable with the Live Typing VM and with the official VM.
- All the configuration of the type information structures should be done in the development environment.
- The programmer should not notice a performance impact, the tools should be usable from the performance point of view.

The rest of the paper presents the Live Typing VM implementation that follows those requirements.

**ACM Reference format:**

Hernán Wilkinson. 2019. VM Support for Live Typing. In *Proceedings of 3nd International Conference on the Art, Science, and Engineering of Programming (<Programming'19> Companion). ACM, New York, NY, USA, 2 pages.* https://doi.org/xxx

## 2 IMPLEMENTATION

In this section we will see the changes made to the VM in order to support Live Typing.

### 2.1 Types Array

The classes of the assigned/returned objects are stored in an array instantiated per variable/method return and accessible by the IDE tools and VM. Its dimension can be configured, being 10 by default. An array is used because it is the simplest collection and it is well known by the VM.

When a class has to be stored in the Types Array, the VM looks for the first empty position of it, that is a position that points to *nil*, and stores the class there. If while looking for that position there is one that refers to the same class, the VM stops looking because it means that the type was already stored. If the end of the

array is reached the class is discarded and that information is lost. If the Types Array is *nil*, it means that no class has to be store for that particular variable/method return.

Figure 1 shows the pseudo code of the algorithm that stores the object's class in the Types Array.

```
keepTypeInformation(typesArray, assignedObject) {
   if(typesArray==nil) return;
   assignedObjectClass = objectMemory.classOf(assignedObject);
   typesArraySize = objectMemory.lengthOf(typesArray);
   for(index=0;index++;index<typesArraySize) {
      if(typesArray[index]==assignedObjectClass) return;
      if(typesArray[index]==nil) {
          typesArray[index] = assginedObjectClass;
          return; }}}
```

**Figure 1: Pseudo code for adding a class to the Types Array**

## 2.2 Instance Variables Types

To store the classes of instance variables, we use the fact that they are numbered in Smalltalk as they appear in the class definition, from 0 to the number of instance variables minus 1.

A new instance variable called *instanceVariablesRawTypes*, was added to *ClassDescription*, a class that defines part of the structure and behavior of classes and meta-classes (classes are objects in Smalltalk). *instanceVariablesRawTypes* points to an array whose size is equal to the number of instance variables and each element of it points to a Types Array. Therefore, when an object is assigned to an instance variable, its index is used to get the Types Array from *instanceVariablesRawTypes*, where the object's class will be finally added.

The bytecodes *storeAndPopReceiverVariableBytecode* and *extendedStoreBytecodePop* were modified to store the classes of instance variables.

## 2.3 Methods Parameters and Temporaries Types

To keep method's parameters and temporaries types, an instance variable called *variablesRawTypes* was added to *AdditionalMethodState*, a class used to store method properties and pragmas. *variablesRawTypes* is used in the same way as *instanceVariablesRawTypes* is used for instance variables because parameters and temporaries are also numbered as they are defined in the method's source code. The variable's index is therefore used to get the Types Array where the class of the assigned object will be stored.

The bytecodes *storeAndPopTemporaryVariableBytecode* and *extendedStoreBytecodePop* were modified to store the classes of temporary variables. To store the classes of the parameters the method activation code was changed because parameters are push on the execution stack.

## 2.4 Method Return Types

The classes of returned objects are stored in a Types Array pointed by an instance variable called *returnRawTypes* added to *AdditionalMethodState*.

# 3 BENCHMARKS

The performance impact average is 1.4x. This value is the average of the time it took to run different test suites in the Live Typing VM and the official VM. The worst case scenario was 1.6x and the best case 1.08x.

The performance impact goes unnoticed in the normal use of the development environment making Live Typing completely usable, although the programmer will notice the difference when executing "long" tasks like loading code or running long test suites.

Regarding memory usage, with a full deployment of Live Typing for all classes and methods in a typical Cuis development environment, its size grows from 10 MB to 17 MB.

A typical Cuis development environment has 792 classes (which doubles to 1584 if we count meta-classes) with a total of 1754 instance variables and 19702 methods that totals 23640 parameters and temporaries (not counting closures parameters and temporaries).

# 4 USAGE

Many tools have been developed or improved in Cuis that exploit Live Typing:

1. Looking for senders and implementors of a message based on the type of the message's receiver. (actual senders/implementors)
2. A new tool that shows the type information of the AST node where the editor's cursor is located.
3. The message rename refactoring was improved to be applied only to the actual implementors and senders regarding the type of the receiver of the message being renamed.
4. The autocompletion uses Live Typing to propose messages in static contexts such as the code editor (class browser).

# 5 FUTURE WORK

The following features are being developed or will be in the near future:

1. Support for closure's parameters and temporaries type info (currently under development).

2. Support for Generics. Needed for collections or classes such as Association (currently under development).
3. Type checker based on Live Typing.
4. Syntax highlighting based on the Type checker of the previous point.
5. Improvement of message related refactorings (add parameter, remove parameter, etc.).

## 6 RELATED WORK

There are many different techniques that generate type information in Dynamically Typed Languages such as type inference [10,11,12,13], manual type annotation [16,17], polymorphic inline cache [14,15] and type profiling [18].

Type inference and manual type annotation focus on providing type information to the development tools as live typing does, PIC focus in performance improvement while type profiling in both.

Type inference predicts the types of variables/method returns while live typing annotates "real" types based on the running code.

Manual type annotation is based on human intervention while Live Typing is not. With manual type annotation the programmer can decide not to annotate a variable or forget to do it, and it is the programmer's responsibility to maintain the annotations. With Live Typing all variables are annotated and the only needed programmer intervention is when objects of previous annotated types are not assigned to a variable anymore. Manual type annotation can annotate types on code that is not run while Live Typing will not provide any type information at all on not run code.

Regarding PIC, its goal is different from the Live Typing's one. PIC's goal is to improve performance while Live Typing's goal is to improve the user experience at the expense of performance as mentioned in section 3. Live Typing could be used to improve performance in a similar way that PIC does, but that remains an open topic.

Type profiling shows type information only at debug time while Live Typing does it at debug and development time. Its implementation adds special byte codes to a method to profile the types while Live Typing modifies existing VM's byte codes keeping unchanged the byte codes generated by the compiler.

## 7 CONCLUSION

Live Typing is a simple implementation that provides usable type information for dynamically typed languages. Development tools and user experience can be greatly improved with Live Typing resembling the functionality of static typed languages tools.

Performance impact is in average 1.4x and therefore imperceptible to the programmer in most of the tasks. Memory size impact is 1.9x in a full deployment of Live Typing but can be configured to be less.

## ACKNOWLEDGMENTS

The implementation of Live Typing is being sponsored by 10Pines SRL. I want to thank Juan Vuletich, Máximo Prieto, Nahuel Garbezza, Nicolás Papagna, and Facundo Gelatti for their suggestions and feedback on the development of Live Typing. I also like to thank Gilad Bracha for his suggestions and support, Eliot Miranda for his helps with the OpenSmalltalk-VM, Tudor Girba for his feedback during the Smalltalks 2018 presentation, and the Smalltalk team at 10Pines.

## REFERENCES

[1] Hernán Wilkinson. 2018. Live Typing: https://github.com/hernanwilkinson/LiveTyping
[2] Juan Vuletich. 2008. Cuis: https://github.com/Cuis-Smalltalk/Cuis-Smalltalk-Dev
[3] S. Ducasse, D. Zagidulin, N. Hess, D. Chloupis. *Pharo by Example 5*. 2018
[4] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay. 1997. *Back to the future: the story of Squeak, a practical Smalltalk written in itself*. OOPSLA '97 Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. Pages 318-326
[5] A. Goldberg, D. Robson, M. Harrison. 1983. *Smalltalk-80: The language and its Implementation*. Addison-Wesley.
[6] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. Commun. ACM, 3(4):184–195, April 1960. ISSN 0001-0782. doi: 10.1145/367177.367199.
[7] John C. Reynolds. Definitional interpreters for higher-order programming languages. In Proceedings of the ACM Annual Conference - Volume 2, ACM'72, pages 717–740, New York, NY, USA, 1972. ACM. doi: 10.1145/800194.805852
[8] Ian Piumarta. Open, extensible composition models. In Proceedings of the 1st International Workshop on Free Composition, FREECO '11, pages 2:1–2:5, New York, NY, USA, 2011a. ACM. ISBN 978-1-4503-0892-2. doi: 10.1145/2068776.2068778
[9] OpenSmalltalk-VM: https://github.com/OpenSmalltalk/opensmalltalk-vm.
[10] Norihisa Suzuki. Inferring types in Smalltalk. In Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. Pages 187-199 . Williamsburg, Virginia — January 26 - 28, 1981 ACM New York, NY, USA. ISBN:0-89791-029-X doi:10.1145/567532.567553
[11] Alan H. Borning and Dan H. Ingalls. A Type Declaration and Inference System for Smalltalk, Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, pp. 133-141, Albuquerque, New Mexico, January 1982
[12] J. Eifrig, S. Smith, V. Trifonov. Sound Polymorphic Type Inference for Objects, OOPSLA'95 Object-Oriented Programming Systems, Languages and Applications, pp. 169-184, 1995
[13] Justin O. Graver. Type-Checking and Type-Inference for Object-Oriented Programming Languages, Ph.D. thesis, University of Illinois at Urbana-Champaign, August 1989
[14] Hölzle, U., Chambers, C., AND Ungar, D. 1991. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In Proceedings of the ECOOP '91 Conference. Lecture Notes in Computer Science, vol. 512. Springer-Verlag, Berlin
[15] Nevena Milojković, Clément Béra, Mohammad Ghafari, Oscar Nierstrasz. Inferring Types by Mining Class Usage Frequency from Inline Caches. Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies Article No. 6, Prague, Czech Republic — August 23 - 24, 2016 ACM New York, NY, USA. ISBN: 978-1-4503-4524-8 doi:10.1145/2991041.2991047.
[16] Gilad Bracha, David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environments. Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) 1993
[17] TypeScript: https://www.typescriptlang.org/
[18] Type Profiling: https://webkit.org/blog/3846/type-profiling-and-code-coverage-profiling-for-javascript/