Journal of the Arkansas Academy of Science

Volume 73 Article 12

2019

Analyzing the Adoption Rate of Local Variable Type Inference in Open-source Java 10 Projects

Clayton Liddell
Arkansas State University, cbwliddell@gmail.com

Donghoon Kim
Arkansas State University, dhkim@astate.edu

Analyzing the Adoption Rate of Local Variable Type Inference in Open-source Java 10 Projects

C.B. Liddell and D. Kim*

Department of Computer Science, Arkansas State University, Jonesboro, AR, USA

*Correspondence: dhkim@astate.edu

Running Title: Analyzing the Adoption Rate of LVTI in Java 10 Projects

Abstract

Type Inference is used in programming languages to improve writability. In this paper, we will be looking more specifically at Local Variable Type Inference (LVTI). For those unfamiliar with LVTI, we will also give an in-depth explanation of what it is and how it works. There is a lot of debate surrounding Type Inference in modern day programming languages. More specifically, whether the costs associated with LVTI outweigh the benefits. It has found its way into many higher-level languages including C#, C++, JavaScript, Swift, Kotlin, Rust, Go, etc. In this paper, we will look at the usefulness of LVTI and its popularity since the release of Java 10. Our study will show that LVTI in Java has not received widespread adoption. We will also explain a possible reason for this, based on the information we have gather from our empirical study which involved statically analyzing 6 popular open source Java 10 projects. We will also discuss different scenarios in which Type Inference can obscure different programming errors.

Introduction

Type Inference is a programming language feature that allows for compilers to, based on the context of a given procedure in a programming language, infer the type of a l-value or a r-value (Agarwal and Stoller 2004). Local Variable Type Inference (LVTI) is Type Inference restricted to the local scope of a program and only used for inferring the type of variables. In Java 10, a new LVTI operator was introduced which is the `var` operator. In order for it to be used, one simply uses the keyword `var` in the place of where the type would be specified. For an example of this, please refer to Figure 3. This allows for programmers to speed up their development process by not memorizing complex datatypes returned by different functions and operators and instead just use the `var` keyword. It can even be used in the place of basic datatypes such as int, char, float, etc. This helps improve the writability of a developer's code while programming, and dramatically increase the speed of the development process. There are, however, some drawbacks that come with LVTI that are most closely related to the readability of a programmer's code who uses LVTI. LVTI can lead to errors that are very difficult to debug as shown in Figure 4, and code that is very difficult to read if not well documented. One popular saying recalled by R. C. Martin in his book *A handbook of agile software craftsmanship* is "Code is read more than it is written", ergo a programs readability is far more important than it's writability in most cases (Danial 2018).

Though this debate has been around for as long as Type Inference has been around, there has yet to be studies performed on LVTI in the newly released Java 10. Additionally, due to Java's Eclipse IDE (Integrated Development Environment), programmers may find the readability of code using the `var` keyword may differ from most other static programming languages which lack a native IDE.

In order to obtain a greater understanding of the true popularity of Java 10's new LVTI feature, we turn to the true judge on the issue: the programming code of the greater Java community. In this project we statically analyzed the frequency of the LVTI feature in 6 popular opensource Java 10 projects.

The results of our study showed that the frequency of usage of Java 10's LVTI is very low to non-existent in most Java 10 projects. This may be due to just how new Java 10 is. Additionally, most of the projects we analyzed have been around for a few years at the least and are currently working on ensuring their projects compatibility with Java 10 before using some of its newer features. It could also be related to developers not wanting to use LVTI due to its drawbacks.

The following contributions will be made through this paper: An empirical study of Java 10's LVTI in real world open source projects.

- A discussion of why Java 10's LVTI has not reached widespread adoption.

- A discussion of the benefits and hinderances associated with LVTI in Java 10.
- An in depth look at different bugs and errors that may be caused by Java's LVTI.

Related Works

A number of articles have been written relating to the adoption of a new programming language feature. There are many static analysis tools for open source projects (Beller *et al.* 2016; Hellström 2009). Kim et al. conducted analyzing type inference in C# (Kim *et al.* 2013) with static analysis tool for open source projects. They showed the usage of `var` type with the number of developers in each open source project. Kim and Yi conducted the acceptance of programming language features commonly referred to as "syntactic sugars" (Kim and Yi 2014). They examined the acceptance of different features that had been around for quite some time in both C# and Java. But they did not examine Type Inference in particular within either of the languages. Smith and Cartwright proposed type inference algorithm that can calculate correct results because Java 5 algorithm fails (Smith and Cartwright 2008).

Background

There have been a few different versions of Type Inference released in Java and changes that have been made to each of these since Java 5. From the perspective of an outsider who is un-familiar with Java or a new Java 10 developer without past experience using Java, LVTI may not seem to be anything special. A feature common to most statically typed programming languages to help with especially complex data-types. However, to a veteran Java developer, LVTI is a game changer. Unlike previous versions of Type Inference features in Java, it infers the entire type of a variable, not just parameter types or generic types. To some developers, this seems like a very nice feature to have and makes the development process much faster and easier. However, to others it may seem like a cause for poorly written, hard to debug code. There is no performance difference between explicit type declarations and implicit ones because `var` keyword instructs the compiler to infer the exact type from the right side of the initialization statement at compile-time based on the type inference algorithm (Kim *et al.* 2013; Agarwal and Stoller 2004).

Knowing the significant impact that LVTI would on the Java ecosystem, we constructed 3 research questions that we sought to answer:

- Is LVTI used widely in Java 10 projects? Why? Or why not?
- Why was LVTI added to Java?
- Are LVTI related errors hard to debug? Why? Or why not?

Implementation

We analyzed 6 open source projects to answer the research questions we discussed within the background section. We selected projects that were within the Java 10 category, according to Github.com. Table 1 displays the name of each project, the lines of Java code within the project, and the total lines of code in the project measured by the opensource command line tool CLOC (Pierce and Turner 2000).

We modified an existing open source programming language analysis framework, written in Java, Python, and MySQL (Parnin *et al.* 2011). The framework statically analyzes code in the following steps:

1. Download the full history of each project from a remote git repository using the git command line tool
2. Store the different file revisions in an intermediate format
3. Transfer the information about each revision to a table within a database
4. Extract occurrences of LVTI from each file revision and store the occurrence in an intermediate format.
5. Store the number of occurrences in the database.
6. Generate graphs for each project analyzed using Octave.

Evaluation

Research Question 1: Is LVTI used widely in Java 10 projects? Why? Or why not?

Our first research question is whether LVTI is used frequently in opensource Java 10 projects. To answer

Table 1: The 6 open source Java projects under investigation

Project Name	Lines of Java Code	Total Lines of Code
d3x-morpheus	65,854	92,104
PMD	116,597	271,882
Jenkins	154,660	250,658
Netty	259,740	270,643
Kafka	228,655	349,444
Elastic Search	1,256,251	1,516,724

this we measured the number of occurrences of LVTI in the opensource projects we downloaded. Figure 1 shows a side-by-side comparison of the usage of LVTI in each of the projects we analyzed. Only two out of the six projects we analyzed had any occurrences of the *var* keyword leaving four projects not using the newly introduced feature at all. Figure 2 shows the number of instances of `var` in each project over time. One of the projects (d3x-morpheus) showed a complete conversion of regular typing to the `var` keyword where possible in the matter of a very short time as can be seen in Figure 2. This project turned out to have the greatest number of occurrences with a staggering 142 instances of LVTI. It can be noted that most of the Java 10 projects analyzed have yet to use LVTI at all.

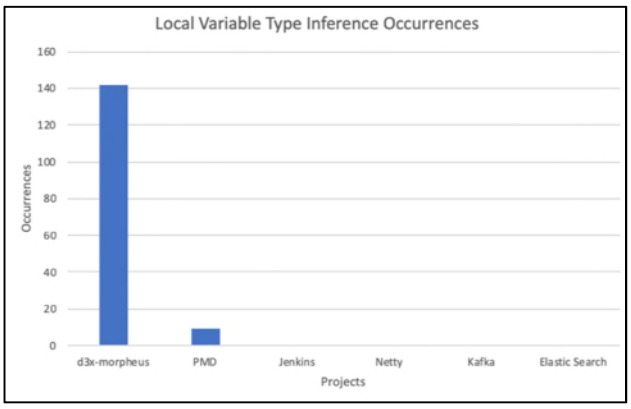

Figure 1: The occurrences of LVTI in the Java projects we analyzed.

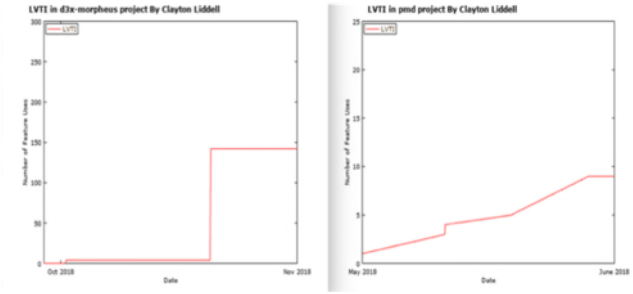

Figure 2: The occurrences of `var` in the two Java projects which used LVTI.

Research Question 2: Why was LVTI added to Java?

LVTI was added to Java in order to increase its writability as a programming language. It supports implicit typing of local variables, which offers succinct syntax, compared with explicitly typed variables (Goetz B. 2018; Marks S. 2018). In C#, the usage of implicit generics declaration (i.e., `var`) is relatively low and a small number of developers use `var`. However, developers would be more likely to use `var` when creating a long variable name with several parameters (Kim *et al.* 2013). It allows developers to not worry about keeping track of the types returned by different functions and dramatically speeds up the development process. It also helps decrease the redundancy of some code, such as when you create an object of a certain type and then have to re-specify the type for the l-value. This can be seen in Figure 3.

```
FileWriter writer = new FileWriter(file);
//becomes
var writer = new FileWriter(file);
```

Figure 3: Example usage of LVTI in Java 10.

Research Question 3: Are LVTI related errors hard to debug? Why? Or why not?

Using Type Inference can make code more concise. But, in some case, use of Type Inference can harm the readability of the program. Because the lack of a type being explicitly declared may make the code harder to read. An example of this can be seen in Figure 4.

```
// What does abd() return?
      var x = abc();

// The type returned from abc() is String!
      String x = abc()
```

Figure 4: The return type of `abc()` is unclear and so the type of x is also unclear.

As you can see in Figure 4, the use of explicit typing makes the code much easier to understand for someone new to a project. If it were not for explicit typing, a person new to the project may have to track down where the function `abc()` is declared just to determine its type.

This may not actually be a problem for someone who uses the Java Eclipse IDE though, since upon hovering over a function, the return type of that function is revealed to the user. So, in this case, the answer is both yes and no. Type Inference can make type related errors harder to debug, however the likelihood of type related errors due to LVTI is mitigated by a feature of the Java Eclipse IDE.

There is another case of an error being caused by Java's LVTI in conjunction with Polymorphism. This issue is addressed in Pierce and Turner's *Local Type*

Inference, however they never arrived at a solution to the issue. The issue is as follows: Imagine you have a parent class Vehicle created alongside child classes Bike and Car. If a variable is initialized with one of the child classes Bike or Car using Type Inference, it cannot be change to the other. Of course, this is an issue that arises in Polymorphism alone even without Type Inference and is often labeled as a feature rather than an issue. However, not being able to assign a variable to a child class of the same parent as the type the variable was first initialized with can be very confusing for an amateur programmer just starting out. And, the `var` operator only adds to the confusion if the programmer is mistaken on what type the variable was first initialized with. This can lead to a very difficult to debug error. For an example of this, see Figure 5.

```
class Vehicle {}
class Bike extends Vehicle {}
class Car extends Vehicle {}

var v = Bike();
//several lines of code later
v = Car();
```

Figure 5: Will result in an error due to incompatible types.

So, while LVTI in Java can make some errors hard to debug it's only in some uncommon cases.

Conclusion

We analyzed the usage of LVTI in 6 open source Java 10 projects and found that LVTI in Java 10 has yet to reach widespread adoption. It is a very handy feature that improves Java's writability with some drawbacks and seems to be a nice addition to Java 10. However, it is still too soon to tell whether it will become widely accepted within the Java community. Some reasons behind why it has yet to reach widespread adoption include: (1) LVTI is so new, that people have not had time to start using it. And, open source developers are still working to make sure their projects are Java 10 compatible, and (2) Java developers may be dissuaded from using LVTI due to the drawbacks we discussed.

We also discussed some Type Inference related errors and how LVTI can harm Java's readability and can also result in errors relating to incompatible typing in Java being more difficult to identify.

Future work may be needed at a later date in order to determine whether Java 10's LVTI becomes widely adopted given time.

Literature Cited

Agarwal R and **SD Stoller.** 2004, January. Type inference for parameterized race-free Java. International Workshop on Verification, Model Checking, and Abstract Interpretation (Springer, Berlin, Heidelberg). p 149-60.

Beller M, R Bholanath, S McIntosh, and **A Zaidman.** 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER) 1:470-481.

Danial A. 2018. AlDanial/cloc. (October 2018). Retrieved November 02, 2018 from https://github.com/AlDanial/cloc

Goetz B. 2018. JEP 286: Local-Variable Type Inference. <https://openjdk.java.net/jeps/286> Accessed on Aug. 27 2019

Hellström P. 2009. Tools for static code analysis: A survey [(MS thesis]. Linkopings, Sweden Linkopings Universitat. 118 p. Available at diva-portal.org.

Kim D and **G Yi.** 2014. Measuring Syntactic Sugar Usage in Programming Languages: An Empirical Study of C# and Java Projects. *In*: Jeong H., M. Obaidat, N. Yen, J. Park, editors. Advances in Computer Science and its Applications. Lecture Notes in Electrical Engineering, vol 279. Springer (Berlin, Heidelberg), p279–284.

Kim D, ER Murphy-Hill, C Parnin, C Bird, and **R Garcia.** 2013. The Reaction of Open-Source Projects to New Language Features: An Empirical Study of C# Generics. Journal of Object Technology 12(4): 1-1.

Marks S. 2018. Style Guidelines for Local Variable Type Inference in Java, < https://openjdk.java.net/projects/amber/LVTIstyle.html> Accessed on Aug. 27 2019.

Martin RC. 2009. Clean code: a handbook of agile software craftsmanship. Prentice Hall (U.S.A). 464 p.

Parnin C, C Bird, and **E Murphy-Hill.** 2011. Java generics adoption. Proceeding of the 8th working conference on Mining software repositories. MSR 11: 3-12.

Pierce BC and **DN Turner.** 2000. Local type inference. ACM Transactions on Programming Languages and Systems 22(1): 1–44.

Smith D and **R Cartwright.** 2008. Java type inference is broken: can we fix it? ACM Sigplan Notices 43(10): 505-524.