Type inference in Dart

The Minimal Dart Checker

Malte Johansson
LTH
tfy11mj2@student.lu.se

Mikel Lindholm
LTH
dat12ml1@student.lu.se

Abstract

This report describes the implementation of a type inference tool called Minimal Dart Checker for the programming language Dart. The tool helps the programmer to statically check the runtime compatibility of types for variables, including type inference with control flow analysis. The tool was built using JastAdd[1].

1. Introduction

Dart is a programming language developed by Google and first released in late 2013[2]. The language uses a dynamic type system and optional types[3] but the official tools offers little support in terms of static type checking. The available static type checker called Dart static checker[4] is written with the intention to only provide warnings for type errors the static checker deems serious and ignore the rest[4]. Our aim is to develop a more strict static type checker for a subset of Dart that produces warnings for more cases of type errors.

Our tool called Minimal Dart Checker is based on type inference. Type inference refers to static deduction of types by analyzing the source code.

2. Dart

Google's Language Dart is a programming language that comes with it's own Virtual Machine (VM). VM is a managed runtime platform on the computer. Running code on a VM is simple, because the emulated machine is a specific computer system which means the code does not have to be recompiled for each computer, but rather compiled only once for the emulated machine. In this sense, a VM is rather versatile, and acts as an abstract layer between the real machine and the emulated machine.

Dart uses optional and dynamic types, meaning that the programmer can choose to specify the type of variables and parameters or leave it as dynamic. Types can be checked at runtime and wrong type assignments will therefore not cause compilation error. However, using wrong types may or may not cause runtime error, this is because Dart is also a weakly typed language and using incorrect types may not be an error, but rather be intended by the programmer. Dart tries to give the programmer the full benefit of dynamic typing, but offers dynamic type checking in *checked mode* in the VM and some static type checking to help avoid incorrect assignment. The effect is that we can, for example, write

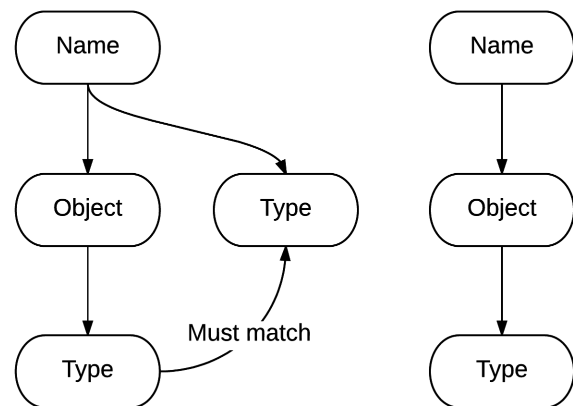
```
int a = 9;
var b = "Foo Bar";
a = b;
print(a);

Output:
$ Foo Bar
```

In the example above a value of type `String` is assigned to a variable with the declared type `int`. Printing the variable after the assignment shows that it contains a value of type `String`.

This may seem unintuitive for a programmer with an object oriented background because in most object oriented languages the type of the variable constrains the variable to only hold values with a compatible type, but the above example is valid for both compilation and runtime in Dart, if run in *production mode* in the Dart VM. This is the power of dynamically typed languages like Dart[5], Python[6], Common Lisp[7], etc. Variable names are not bound to an object and a type, but only to the object. Types come from the objects, not from the variable. This is illustrated in figure 1 below.

Figure 1. The figure illustrates how types work for different languages. To the left is an illustration of how types are bound to both the variable and the object, for statically typed languages. This can be compared to the right illustration, which shows how types are only bound to the variable for dynamically typed languages.



Therefore we can write as in the code example above, where *a* is declared as an integer, but this is disregarded both in compilation and execution in production mode in the Dart VM because the objects hold the types and not the variables. Declaring types can be seen as redundant, because when programs are run in the default production mode, which does not do any type checking, it treats all variables as dynamic. Therefore the following code is valid in production mode, but not in checked mode. The code fails in checked mode because, as seen in the warning message, a `bool` type is assigned to a variable declared as a `String`.

```
String i = "Foo Bar";
i = 10 > 5; // raises error in checked mode.
```

```
print(i);

Warnings:
$ type 'bool' is not a subtype of type 'String' of
      'i'.

Output:
$ true
```

Programs that run in production mode are faster, because the Dart VM can avoid extra type checking. This can of course lead to problems if the programmer makes a mistake. Checked mode can then be used, where the types are actually checked and the VM raises an error during runtime, terminating the program with an error message. Google has developed an in-browser IDE called DartPad[8], which provides a static type checker. Dart's Static type checker only gives warnings when an assignment may lead to an error, much like lint in C. Dart does not have coercion, i.e. implicit type conversion, but instead has assignment compatibility which is bidirectional. Assigning a subtype to a supertype is okay in most statically typed languages, but Dart's assignment compatibility is bidirectional and it is therefore not a problem to assign a super type to a subtype. To demonstrate this aspect of Dart, consider the following example. In the example, a supertype `Vehicle` is assigned to a subtype `car` and the subtype `Ambulance` is assigned to the supertype `Vehicle`. This shows that Dart's assignment compatibility system is bidirectional.

```
class Vehicle { String sound() => '*No sound*'; }
class Car extends Vehicle {
    String honk() => 'Toot'; }
class Ambulance extends Vehicle {
    String siren() => 'Wioo wioo'; }

main(){
  Vehicle vehicle = new Vehicle();
  Car car = new Car();
  Ambulance ambulance = new Ambulance();

  car = vehicle;
  print(car.sound());

  vehicle = ambulance;
  print(vehicle.siren());
}

Output:
$ *No sound*
$ Wioo wioo
```

In many object oriented programming languages like Java[9] and C# [10], this would not be legal since the compiler cannot be sure that all functions called on a subtype can be found in the supertype, e.g. the 'siren()' method in the class `Ambulance`. Using assignment compatibility which supports bidirectional assignments however, like Dart does, the example is valid. There are limitations of course, for example calling the method 'honk()' on an instance of Vehicle would throw the runtime error '*Uncaught TypeError: Vehicle.honk$0 is not a function*'.

To summarize, DartPad uses a static type checker which is a lot like a lint tool and warns about some assignments that do not fit Dart's assignment compatibility rules. The code can still be valid and run in production mode in the Dart VM, however it will not pass in checked mode because then the Dart VM actually does type checking.

3. Dart type checking limitations

When looking at Dart's own tool for type checking it can be good to have an example to see exactly what problems there are and also what is done to warn about these. Therefore, consider the following example which we will use in the rest of this section.

Code example 1. Dart example

```
var v = 8;
bool b = "ABC"; // warning 1
b = v; // warning 2

if(b)
{
  v = "ABC";
} else {
  v = 1;
}
b = v; // no warning
print(b);

Warnings:
$ A value of type 'String' cannot be assigned to a
     variable of type 'bool'
$ A value of type 'int' cannot be assigned to a
    variable of type 'bool'
Output:
$ ABC
```

The example starts with a couple of assignments, where the second and third assignments contain incorrect type assignments. Assignment two tries to assign a variable of type `bool` to a value of type `String`, which is not allowed. Assignment three assigns a `bool` typed variable to a value of type `int`. This is, like the previous assignment, not allowed because of the mismatch of types. Next is an `if-else` statement. The `if` branch contains an assignment to variable v with a value of type `String` and the `else` branch contains an assignment to variable v with a value of type `int`. The next assignment is now an incorrect type assignment since b is a variable of type `bool` and v is assigned to either a value of type `String` or `int`. The warning messages are generated in real-time with the Dart static checker in Dartpad.

3.1 Dart Static Checker

3.1.1 Control flow

Consider the code above. Dart's static checker is able to perform type checking for the first assignment and is able to infer the type of the variable v for the second assignment. However, it is unable to do so after the `if` statement, where the variable v can have the type `int` or `String`, depending on which branch is executed. When Dart's static checker encounters branching, as with the `if` statement, it simply drops the warning because it can not infer the type.

The Dart static checker seems to be unable to provide warnings when a variable can have multiple types even though the desired type is not included in the possible types. As in the example above, where the last assignment tries to give a variable of type `bool` the value of a variable which can only have either the type `String` or the type `int`. This is a drawback, since every branch is not checked it can lead to a hard time finding the bugs or even errors since the programmer is not warned that a variable may contain a value of an unexpected type.

3.1.2 Return type checking

One of the bigger limitations for Darts static checker is whenever a function is called, the return type is never inferred but rather

statically checked. This mean that the below code passes the static checker, except for the last assignment in the main function, but when run in Dart VM's checked mode it will get a runtime error instantaneously at the second assignment, due to a mismatch in types. This limitation is a big inconvenience due to the amount of assignments of function calls a normal program may contain. If the return values are not checked, then there might be an error at each assignment of a function call.

```
main(){
  dynamic d = "Foo";
  bool b = func(); // no warning
  b=d; // warning
}

dynamic func() {
  return "Bar";
}

Warnings:
$ A value of type 'String' cannot be assigned to a
      variable of type 'bool'
```

The problem of inferring types does not seem consistent either. As we saw in the example code abode, the static checker issues a warning for the last assignment in the main function. The static checker seems happy to infer a `String` type for the variable d. When using a function call to assign the variable b, of type bool, it does not care about what the function actually returns, even though the function is statically declared to return a value with the type `String`.

3.2 Dart checked mode

Some type errors can be caught by Dart's static checker, but as we saw in the previous section, not all type errors generate warnings. In fact, the example 1 in section 3, will only get two warnings from the static checker and will run fine in the Dart VM's production mode. Despite being able to compile and execute, the code will generate a runtime error when run in checked mode. Removing the error, compiling and then running the code again shows the next error. This process has to be repeated for all errors in the code that the checked mode can catch. The process of removing bugs from a program can be perceived as tedious since only one error message is shown at a time. The shortened printout below shows the three errors that the code in the Dart example 1 produces, from each iteration of the process just described.

```
Error 1:
 Unhandled exception:
 type 'String' is not a subtype of type 'bool' of
      'b'.

Error 2:
 Unhandled exception:
 type 'int' is not a subtype of type 'bool' of 'b
      '.

Error 3:
 Unhandled exception:
 type 'String' is not a subtype of type 'bool' of
      'b'.
```

This is expected since Dart VM's checked mode actually runs the code and terminates with a warning when a type error occurs. The disadvantage is that only one branch being executed is tested and that this requires the code to be compiled and run. This makes Dart VM's checked mode a poor choice if the user wants to test the correctness of different branches with different values.

4. Type inference used by the Minimal Dart Checker

The Minimal Dart Checker is a type inference tool that uses type inference to deduce the type of a variable which allows the tool to perform more accurate type checking. In the example below no warning is generated since type inference is used on line 2 to deduce that the variable v with the declared type `var` actually contains a value of type `int`.

```
var v = 1;
int i = v;
```

Minimal Dart Checker also features control flow type inference and return type inference, described in section 8.

5. Type systems in other languages

In this section we investigate type systems in other languages.

5.1 Type inference in Haskell

Haskell is a statically typed language which means that all variables are assigned a type before runtime. This is made possible by type inference [11] [12]. This is similar to the static type analysis in the `Minimal Dart Checker` where it attempts to deduce the type for each usage of a variable using type inference. The difference is that a variable in Dart can hold values of different types, which makes it inadequate to simply deduce the type of a variable once. The Dart code below shows an example of this.

```
var v = 1;
int i = v;
v = "string";
String s = v;
```

In Haskell, functions and values can be type specified which means that the programmer tells the compiler what types to expect and use. The compiler checks of course that it can infer the type that was specified. If no type was specified, the compiler then infers the types itself if it can be done. Compilation errors occur when the compiler can not infer the same type as specified, or when value types and the function arguments do not match.

5.2 Type hinting in Python

Python is a strongly typed language with a dynamic type system[13]. The language, in version 3.5 [14], offers support for type hinting which is a way for the programmer to hint the type of the value a variable holds. The hints are not however currently used by the language to offer any type checking. Third party tools exist that offers type checking.

This is similar to the optional types in Dart whose purpose is also to make the code more understandable by allowing the programmer to include the actual types and thereby declaring intent. The difference to Dart is that Dart provides both a static checker and a dynamic checker, both developed and provided by Google, that makes use of the optional types to provide some type checking. There exist a third-party IDE for Python, called PyCharm which supports and helps the programmer with type hinting.[15]

6. Implementation

`Minimal Dart Checker` uses JFlex[16] which is a scanner generator to generate a scanner for the subset of Dart that `Minimal Dart Checker` supports. The LALR parser generator Beaver[17] is used to generate a parser for the subset of Dart. JastAdd[1] is then used to implement an attribute grammar that performs type inference. All three components that make up Minimal Dart Checker are built using the language Java.

7. Supported language features

The subset of Dart that the Minimal Dart Checker supports includes the following: Basic Dart statements including assignments, while, return, if and for. Arithmetic, comparative, logical, shifting, increment and decrement operator are also supported. Classes are supported as well as class variables and class functions. Global functions and variables are supported as well. Generics are also supported but type parameters are limited to being non generic. Optional types are also supported.

8. Tool features

Minimal Dart Checker supports type checking without type inference. The Dart code below shows an assignment where type checking can be done without the need of type inference.

```
int i = 1;
```

It also supports type checking with type inference. The Dart code below shows an assignment where type inference is required to investigate the type of `v` which is required for the type checking.

```
var v = 1;
...
int i = v; // type inference is required
```

Minimal Dart Checker supports type inference with control flow analysis. The Dart code below shows an assignment where control flow analysis is required to investigate the type of `v` which is required for the type checking. The resulting error message is displayed below the Dart Code.

```
var v = 1;
v = true;
if(true) {
   v = 2.4;
} else {
   v = "string";
}
String s = v; //type inference with control flow
    analysis is required

Output:
$ Error at line 9: Type 'String' for variable 's'
    does not match '[Double, Bool, String] gives
    inferred type 'Dynamic''
```

Minimal Dart Checker supports type inference with return types. The Dart code below shows a function with a statically declared return type that is `var`. To investigate the actual return type type inference is required. The resulting message is displayed below the Dart code.

```
main() {
       String s = function();
}

var function() {
       if(true) {
              return "ABC";
       } else {
              return 1;
       }

       return 1.1;
}

Output:
$ Error at line 1: Type 'String' for variable 's'
    does not match the types '[Double, Int, String
    ] which gives the inferred type 'Dynamic''
```

Minimal Dart Checker includes a feature that will suggest return type. The Dart code below shows a function with a statically declared return type that does not match the return type being inferred by Minimal Dart Checker. The resulting message is displayed below the Dart code.

```
var function() {

       if(true) {
              return "ABC";
       }else {
              return "DEC";
       }
}

Output:
$ Error at line 1: Function have return type '
    Dynamic' but infers type 'String'
```

9. Discussion

9.1 Too many warnings

Whenever a tool produces a warning for an error pattern that does not definitely lead to the code not working there is a chance that the programmer is aware of the issue and simply chooses to ignore it[18]. It might even be the case that the code that generated a warning behaves as intended. Warnings are after all just a analysis tool complaining that the program does not entirely fulfill every assumption that the creator of the tool has about how a program should work. Writing a analysis tool that complains only about the necessary things is really hard since there are always special cases that might not be caught.

If many warnings are generated by the tool that the programmer does not find useful then this may lead to the programmer either not reading the warnings or disabling them. That is to say that there is an issue with tools that are too liberal with producing warnings. Giving too few warnings can result in buggy programs, and too many can make the programmer not care about the warnings at all and therefore missing the important warnings that will lead to errors at runtime.

The right amount of warnings to report is dependent on the tools purpose and its target audience. What does this mean for a static type inference tool for an language with optional types?

The target audience for such a tool is clearly those who want to use the optional types and want warnings when the types are not followed in an expected way. Therefore we believe that in the context of this tool there should be warnings whenever an assignment is made so that the declared type is inconsistent with its actual type. See an example of this below.

```
int i = "ABC";
```

Here `i` should only be assigned values of type `integers` and not of type `String`.

9.2 Inference with control flow analysis

When it comes to Dart and types, the static checker does not give many warnings about issues that can lead to errors at runtime. This is especially true for control flow, where different branches in the code can lead to different types being assigned to a variable. The static checker gives up as soon as there is a branching statement,

and cannot infer types anymore. Minimal Dart Checker is designed to help infer types even in some of these kinds of situations, as we show below.

Inferring types can be quite ambiguous, and it is often not clear whether the code should give an error or not. See the example below.

```
var v = "Foo Bar";
int i;
if(doSomething())
{
        v = 13;
}

if(doSomethingElse())
{
        i = v; // Should this generate a warning?
}
```

The code above shows an example where it is hard for a static tool to determine the control flow because of lack of information about the functions `doSomething` and `doSomethingElse`. Another reason that it is hard to determine the control flow is that it can depend on input which are not known before execution. In this example a type error will only occur if `doSomething` returns `false` and `doSomethingElse` returns `true`. This might never happen.

We designed Minimal Dart Checker so that the programmer should be aware of any assignments of different types to a variable in the different branches. If the type can be inferred and it is compatible with the assignment, then the tool should not complain. However, if the type can not be inferred, or if the type can be inferred but it is not compatible, then Minimal Dart Checker will issue a warning, listing the different types of the variable that were used. This gives the programmer a heads up of all the different types that were assigned to the variable. That way, the programmers will know if they made a mistake using the wrong variable in one of the branches, or if a different type was assigned that was not intended.

To illustrate the need for the type inference tool consider the example below. Dart's static checker will not complain on the assignment of *v* to *s*. However, Dart's static checker will complain with the message '*A value of type 'bool' cannot be assigned to a variable of type 'String'*' for the assignment of *v2* to the variable *s2*. This illustrates the need for inferred types in a control flow.

```
var v = "Foo Bar";
if(true) {
  v = true;
} else {
  v = 42;
}

String s = v; // No complaint
print(s);

var v2 = true;
String s2 = v2; // Complaint from the static
     checker

Output:
$ 42
```

The error messages from the static checker are not great, it does not even give one for the first assignment after the 'if' statement. In an effort to enlighten the programmer of possible bugs, Minimal Dart Checker gives a warning that more than one type has been assigned to a variable whose inferred type is not compatible. Minimal Dart Checker gives the programmer the necessary information needed: line number, variable name which was assigned, which type it was declared as, the multiple types that were assigned to the other variable and of course which type that was inferred. For example see the Dart code below which contains a control flow problem that results in the messages displayed below the Dart code.

```
num n = 1.3;

for(int i = 1; i < 10; i++) {
        n= 15;
}

int i = n; // warning 1
var v = 1.3;

if(i < 23) {
        v = 21;
        v = true;
} else {
        v = "Bar";
}

bool b = v; // warning 2

Output:
$ Error at line 7: Type 'Int' for variable 'i'
     does not match the inferred type 'Double' from
      the types [Double, Int]'
$ Error at line 17: Type 'Bool' for variable 'b'
     does not match the inferred type 'Dynamic'
     from the types [Double, Bool, String]'
```

9.3 Our Type System

When constructing the AST it is a common pattern to try to attach singular nta nodes for the different types to the root. We wish to do this with the generic classes as well to make code interpretation and comparison between generic types easier. Since it it possible to create generic types with an arbitrary amount of object parameters it is not possible to attach a nta node for every possible type.

The solution is to attach a generic type node for the generic class to the root. And then on demand attach a new node with the specific generic params to the generic class node. This is illustrated in figure 2.

9.3.1 Parsing complications

A problem that emerges when you want to parse a language that contains both shifting with the syntax (">>", "<<") and generics with the syntax "$ClassA < ClassB >$" is to handle the case when the generic parameters are themselves generic.

```
List<List<List<Dog>>> list;
```

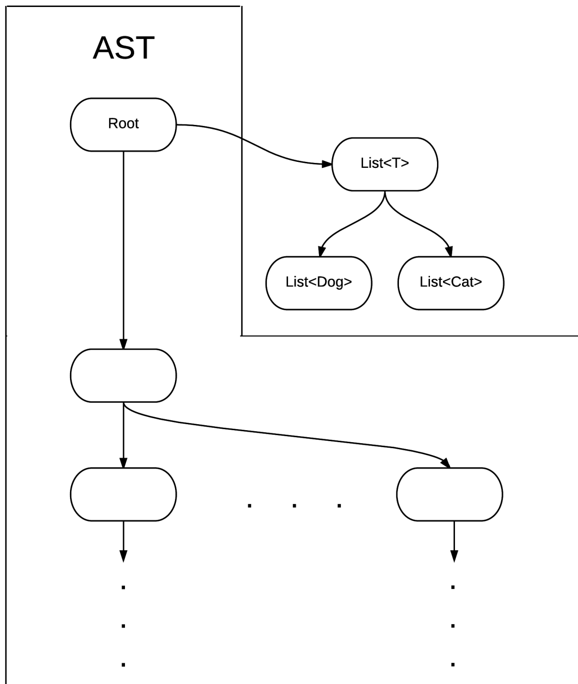
Since the shift token is made up by two characters it will be chosen over the greater or lesser token because of longest match rule in the scanner. This means that depending on the depth of generic arguments the parser should expect different tokens.

This is solved by creating different parsing rules for each depth up until depth of 2 where this ceases to be relevant. For a language with "<<<" the parser would have to have separate parsing rules up until depth 3 instead.

10. Evaluation

In this section we aim to evaluate the `Minimal Dart Checker` inference tool by means of comparing it to other tools available.

Figure 2. Model of the AST that is build with `Minimal Dart Checker`



Two of the most important are of course Dart's own tools, the static checker and the VM's checked mode. Other sections, e.g. 3, have already pointed out some limitations with these two with some examples. Therefore, the focus in this section is to compare features and limitations when comparing other tools to Minimal Dart Checker.

10.1 Comparison with Dart Static Type Checker

As explained in the section 3, the static checker does not work when a variable can contain values of different types because of branching code, the static checker stops inferring the type and leaves the potential bug for the Dart VM's checked mode. This lack of type checking on control flow branches can lead to errors, and the programmer will be none the wiser because the checker does not display any warnings or errors. Minimal Dart Checker offers static analysis over more extensive control flow and gives the programmer a comprehensive set of information in the warning. To show what a 'comprehensive set of information' means, take the example output from Minimal Dart Checker, taken from the last code example in section 9.2. In this warning, the involved variable that was assigned is mentioned, the line at which the warning was generated from, which type was expected and a list of all different types that was found throughout the branches.

```
Error at line 7: Type 'Int' for variable 'i' does
    not match the inferred type 'Double' from the
    types [Double, Int]'
```

In section 3 we looked closer at how Dart 'infers' types of function return values. The static checker seems to neither type check the declared return type nor the inferred return type for a function call. However, Minimal Dart Checker infers the type when the value of a function call is used. Not only that, but it also checks if the return type can be more strict, e.g. if the returning type is always a String then a warning is issued where it is suggested that the return type of the function should be changed to match what is actually returned, `String` in this case.

10.2 Comparison with Dart Checked mode

The advantage of Dart's checked mode is that it does type checks during runtime with the actual runtime variable states. This is however also a disadvantage since it only checks one branch with one set of data and it requires the program to be executed. This leads to potentially poor coverage of the possible type errors that can occur with different control and data flow. This is not the case with Minimal Dart Checker since it collects all the possible types and checks their correctness.

The termination of the program after each type error means that only one error can be found at a time with checked mode. This is not the case with Minimal Dart Checker since it works at a static level and can find all type errors without terminating.

The performance of running a program in checked mode is based on the performance of the program. Searching for errors at the end of a program with long execution times will at the very least take as much time as the execution of the program. This problem is also made more severe since only one error can be found at a time.

Minimal Dart Checker does not have this problem because it is based on static analysis.

10.3 Automatic test suite

With Minimal Dart Checker comes an automatic test suite with 23 test. Nine of these test are focused on parsing the Dart subset and 14 test are focused on type checking.

11. Conclusion

We have found that Minimal Dart Checker is able to produce warnings for error patterns that neither Dart's Static Checker nor Dart's Checked mode can generate warnings for. These error patterns are connected to type checking for optionally typed variables and therefore there is a reason to use Minimal Dart Checker for programmers with the intent to enforce type checking. The limitation of the Minimal Dart Checker is that it only supports a subset of the language and that it does not include Darts official libraries. These two limitation have a large negative impact on the usability of Minimal Dart Checker in the industry since larger Dart projects are likely to make use of Darts official libraries and language aspects not supported by the tool.

References

[1] "Jastadd official homepage." URL:`http://jastadd.org/web/`. retrieved 2016-12-15.

[2] S. Ladd, "Dart news & updates: Dart 1.0: A stable sdk for structured web apps." URL:`http://news.dartlang.org/2013/11/dart-10-stable-sdk-for-structured-web.html`, 2013. retrieved 2015-11-30.

[3] K. Walrath and S. Ladd, *Dart: Up and running*. O'Reilly Media, Inc, 2012.

[4] "Optional types in dart." URL:`https://www.dartlang.org/articles/optional-types/#the-static-checker`. retrieved 2015-11-30.

[5] "Dart official homepage." URL:`https://www.dartlang.org/`. retrieved 2015-12-16.

[6] "Python official homepage." URL:`https://www.python.org/`. retrieved 2015-12-16.

[7] "Common lisp official homepage." URL:`https://common-lisp.net/`. retrieved 2015-12-16.

[8] "Dartpad." URL:`https://dartpad.dartlang.org`. retrieved 2015-12-16.

[9] K. Arnold, J. Gosling, and D. Holmes, *The Java programming language, Fourth Edition*. Addison Wesley Professional, 2005.

[10] "C# official homepage." URL:`https://msdn.microsoft.com/en-us/library/kx37x362.aspx`. retrieved 2015-12-16.

[11] D. Duggan and F. Bent, "Explaining type inference," *Science of Computer Programming*, vol. 27, pp. 37–83, Juli 1996.

[12] M. Lipovaca, *Learn you a Haskell for great good!* No Starch Press, 2011.

[13] "Why is python a dynamic language and also a strongly typed language - python wiki." URL:`https://wiki.python.org/moin/Why%20is%20Python%20a%20dynamic%20language%20and%20also%20a%20strongly%20typed%20language`. retrieved 2015-12-01.

[14] "Pep 0484." URL:`https://www.python.org/dev/peps/pep-0484/`. retrieved 2016-12-01.

[15] Q. N. Islam, *Mastering PyCharm*. Packt Publishing, 2015.

[16] "Jflex official homepage." URL:`http://jflex.de/`. retrieved 2016-12-15.

[17] "Beaver official homepage." URL:`http://beaver.sourceforge.net/`. retrieved 2016-12-15.

[18] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," p. 2, May 2015. International Conference on Software Engineering (ICSE).