Transitive reachability over grammar-guided graphs for static type checking

Chintana Prabhu - 301404072 - cprabhu@sfu.ca   Luiz F. Peres de Oliveira - 301288301 - lperesde@sfu.ca

I. Introduction

Type-safety means that every variable, function argument, and function return value is storing an acceptable kind of data, and the operations that involve values of different types "make sense" and do not cause data loss, incorrect interpretation of bit patterns, or memory corruption. A program that never explicitly or implicitly converts values from one type to another is type-safe by definition, in strongly-typed languages [1]. However, type conversions, even unsafe conversions, are sometimes required. For example, you might have to store the result of a floating point operation in a variable of type **int**, or you might have to pass the value in an **unsigned int** to a function that takes a **signed int**. Both examples illustrate unsafe conversions because they may cause data loss or re-interpretation of a value.

Whenever an unsafe type conversion is detected by a strongly-typed system, either an error or a warning should be issued. An error stops compilation; a warning allows compilation to continue but indicates a possible error in the code, although it still may contain code that leads to implicit type conversions that produce incorrect results. While static casting errors can be caught and avoided in compile-time, the runtime nature of implicit type conversion can be rather problematic, if not done correctly.

Implicit type conversions in strongly-typed systems can be applied to both primitive and reference data types. Apart from data loss and memory corruption, primitive-to-primitive implicit type conversions can be forgiving, and for many incorrect cases, an incorrect program will still run. The same is not true for reference data types. For reference data types, a compiler checks for possible conversions that are specified in the class hierarchy. If the compiler can't find an acceptable conversion, it issues a compile-time error, otherwise it create a compile-time valid dynamic casting operation.

In the dynamic casting operation, it is up to the programmer to make sure type S can be casted into a type T. When types S and T are unrelated, the compiler will raise a dynamic casting error at runtime. Any conversion from one type to another in a dynamic casting can be a potential source of program errors.

This report is focused on static type checking for dynamic casting operations over reference data types in Java programs and is organized as follows: section II explains how dynamic casting operations are handled by the Java compiler and JVM, section III and IV show the motivation of this project as well as related work, section V explains the project layout, architecture and implementation in details, section VI shows our implementation results and finally, section VII concludes our project and discuss future work.

II. Compile-time checking for dynamic operations in the Java compiler and JVM

Java docs [2] show that while reference **upcasting** needs no runtime action, e.g. casting from type *String* to type *Object*, the opposite, **downcasting**, is always true because "a specific conversion from type S to type T allows an expression of type S to be treated as if it had type T instead" and if no checks are performed, a *ClassCastException* could be raised as a result of the unchecked conversion. Suppose we want to cast an instance b from type B to type C which is derived from type A: $A\ a = (C)\ b$. Below we show the rules for dynamic casting used by the Java compiler and JVM:

Compile-time checking - Java Compiler:

- Rule 1: There **must** exist some relation between the type of intance b (B) and C, where either C is a derived from B, B is derived from C or B and C are the same type. If no such relation exists, a compile-time error is raised as the types B and A result into non-convertible types.
- Rule 2: The type being cast (C) **must** be either derived from A or A and B are the same type. If not, a compile-time error is raised as the types of A and C are incompatible.

Runtime checking - JVM:

- Rule 3: At runtime, the type of b (B) must be either the same type of C or derived type from C. Otherwise, a runtime exception of type *ClassCastException* is raised.

As it is evident, one of the major drawbacks in detection of invalid type casting is: it is performed at runtime. It may happen that there exists a part of the code which has been deployed in production and even the testing was not able to detect and rectify the invalid type casting taking place.

In order to help overcome *ClassCastException* errors, Java 5 introduced Generics. Generics provide compile-time checks and can be used to develop safer applications [4]. Apart from Generics, **instanceof** feature could be utilized prior to casting an object to some other type, which tests if an object is instance of a specific *Class*. However, **instanceof** also brings runtime overhead, since we must always check for the object type before every casting operation, and that can be easily forgotten, as programmers might forget to include it before an implicit casting.

III. Motivation

We came across a number of works where advantages of static checking over dynamic checking are discussed. A few of those points caught our attention. Statically-typed languages help in early detection of type errors and usually succeed in providing considerably better runtime performance, in comparison with dynamic typed languages. Statically-typed languages also offer the programmer the benefit of detecting type errors at compile-time making it possible to fix them immediately rather than discovering at runtime, when the programmer's efforts might be aimed at some other task, or even after the program has been deployed [18].

Another comparative study by Okon and Hanenberg et al. experimented to enforce a benefit for dynamically-typed languages in comparison to statically-typed ones [17]. Though their motive was to show that there are measurable benefits for dynamic-typed systems, the experiment showed something different. Only two of four programming tasks revealed a positive impact of using dynamically-typed languages. The other two tasks showed (again) a positive influence of the statically-typed systems.

Keeping these achievements into consideration, we set an objective of our project to prove statically whether or not it is possible to apply dynamic casting, so that runtime errors are avoided - *ClassCastException* in particular. Essentially, our idea is that we can eliminate the runtime overhead that is created when we use **instanceof** so to check the validity of runtime castings as well as catching unchecked castings that might have been done by Java programmers.

IV. Related Work

Most of the attempts made towards having static type checking at compile-time tend to address the costs associated with the dynamic analysis and ways of overcoming it:

- The instrumentation introduces a certain amount of runtime overhead.
- The cost of dynamic typing is largely distributed across the code and process. It tends to slow down the whole process pipeline a little.
- Dynamic typing gets us slow code, where the slowdown depends largely upon the complexity of the type-system.

Yong and Horwitz et al. present their tool RTC(Runtime Type-Checking) which utilizes several techniques for reducing the runtime overhead by using static analysis to identify some cases in which instrumentation can safely be omitted. RTC instruments C programs so that the runtime type of every memory location is tracked during program execution, and inconsistent type uses case warnings and errors to be reported. We summarize the most relevant component of the RTC architecture as follows:

- During the first step, the RTC tool associates with each memory location one of the following runtime types: unallocated, uninitialized, pointer, zero, char, short, int, long, float, double.
- During the second step, the runtime types are stored in a "mirror" of the memory used by the program, with each byte of memory mapped to a four-bit nibble in the mirror (thus incurring a 50% space overhead).
- Finally, it uses Ckit [15] as its front end, and translates a given set of preprocessed C source files into instrumented C files. These are then compiled and linked with the RTC library, producing an executable that performs runtime type checking and reports error and warning messages (most of which were out-of-bound array or pointer accesses).

Padhye and Sen et al. address the problem of the problem of dynamically checking if an instance of class S is also an instance of class T [16]. They propose fail-fast mechanism in this regard. The overall flow of operations of fail-fast are as follows:

- In the compiled version of each class, they store a fixed-width bloom filter, which combines randomly generated type identifiers for all its transitive supertypes.
- At run-time, the bloom filters enable fast refutation of dynamic subtype tests with high probability. If such a refutation cannot be made, the scheme falls back to conventional techniques.
- The scheme stores only one extra machine word per class and requires only a single load + bit-mask test to refute dynamic subtype checks with high probability.
- This scheme works with multiple inheritance, separate compilation, and dynamic class loading. Though their scheme is simply an add-on for existing implementations like HotSpot JVM and LLVM, it attempts to prevent worst-case linear search when it is likely to occur.

Another work, Graspan [5], a single machine, disk-based parallel graph processing system tailored for interprocedural static analyses. It uses an edge-pair centric computation model to compute dynamic transitive closures on very large program graphs. The overall flow of ideas of Graspan is summarized below:

- They are formulating most of the interprocedural analyses problems as a graph reachability problem.
- As an example, they state - in pointer/alias analysis, if an object (e.g., created by a malloc) can directly or transitively reach a variable on a directed graph representation of the program, the variable may point to the object.

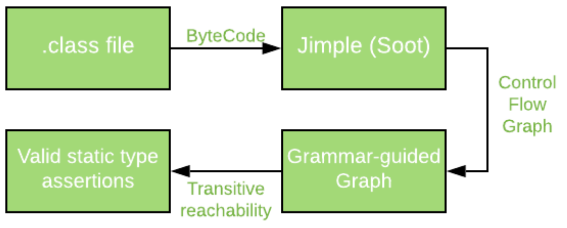
Therefore, they turn the programs into graphs and treat the analyses as a graph traversal. This approach has opened up opportunities to leverage parallel graph processing systems to analyze large programs efficiently. Graspan offers two major performance and scalability benefits: The core computation of the analysis is automatically parallelized and out-of-core support is exploited if the graph is too big to fit in memory.

V. Project Layout and Implementation

Here we present a framework to determine the validity of dynamic casting at compile-time which otherwise would have been performed at runtime. Effectively, we show that is possible to prove the correctness of dynamic casting in Java programs so that they never terminate unexpectedly by *ClassCastException* errors.

In order to accomplish this, we have utilized the concept of grammar-guided transitive reachability on graphs, as an inspiration from the Graspan project [5], a disk-based graph system for interprocedural static analysis of large-scale systems.

Grammar-guided transitive reachability over graphs is a problem where, given a graph $G = (V, E)$, a grammar g and any pair of edges $e_1 = (u, v)$, $e_2 = (v, w)$ in E, transitive edges are created from u to w if and only the labels of edges e_1, e_2 are part of one of the transitive production rules in g. At the end, an extension $G' = (V, E')$ of G is returned. By implementing this technique on a given control flow graph (CFG) of a program and statically analyzing the type conversions occurring in the program, we can find transitive relationships that prove the correctness of dynamic casting operations before an application is sent to production, therefore, reducing unnecessary production bugs and crashes and is divided into 4 phases:



- **.class file**: In the first phase, we compile .java file into a .class file using the java compiler (*javac*), and feed Soot[7] with the .class file (bytecode representation).
- **Jimple (Soot)**: In the second phase of our framework, we use Soot [6] to have access to the control flow graph (CFG) representation of the Java program that was fed as input. The CFG that is provided by Soot is then used to create a grammar-guided graph (G) which is discussed below.
- **Grammar-guided Graph**: In the third phase of our project, the control flow graph is converted into a grammar-guided graph G, which is then extended by new transitive relations that are created through our grammar-guided transitive reachability pass, generating a new graph G'.
- **Static assertion**: At the end, it is possible to prove statically whether or not the types will hold until the end of the application, with no crashes, if G' has the desired target (transitive) edges.

V.I Valid example

The example below gives a detailed demonstration of a valid example that would be handled by our approach.

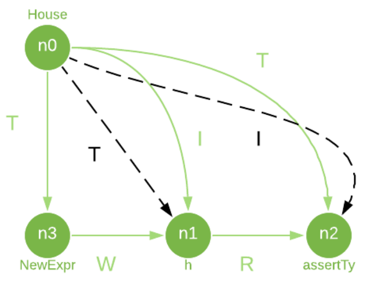
V.I.I Phase 1: Java input

House h = **new House**();
assertTy(h, **House**.class)

V.I.II Phase 2: Simplified Jimple (Soot)

House $\$r2$;
$\$r2$ = **new House**;
specialinvoke $\$r2$.<House: void <init>()>();
staticinvoke <assertTy(obj, class)>($\$r2$, class "LHouse;");

V.I.III Phase 3: Grammar-graph graphs G and G′



Observe that $G = (V, E)$ *and* $G' = (V, E')$*, where* E *is the union of all green edges and* E' *is the union of both green and black edges.*

V.I.IV Phase 4: Static assertion

The main objective in our approach is to have a transitive (black) edge of type **I** (instance) from the target node n_0 to source node n_2, In other words, we want to check whether House is propagated and is one of the valid types for object h until the end of the program. Because this is true for the example above, it is proven that House is a valid type for h and we may remove any **instanceof** checks that might exist around h, therefore reducing the total application overhead.

V.II Invalid example

In the example below, you will see a piece of code which is valid in Java at compile-time, but invalid at runtime and will cause the JVM to terminate the given program due to a *ClassCastException* error.

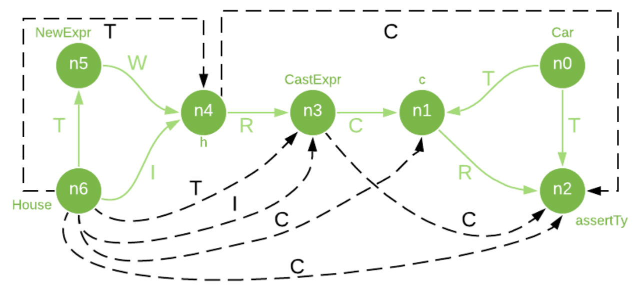
V.II.I Phase 1: Java input

House h = **new House**();
Car c = (**Car**) ((**Object**) h);
assertTy(c, **Car**.class)

V.II.II Phase 2: Simplified Jimple (Soot)

Car $r2$;
House $\$r3$;
$\$r3$ = **new House**;
specialinvoke $\$r3$.<House: void <init>()>();
$r2$ = (**Car**) $\$r3$;
staticinvoke <assertTy(obj, class)>($r2$, class "LCar;");

V.II.III Phase 3: Grammar-graph graphs G and G'



Observe that $G = (V, E)$ and $G' = (V, E')$, where E is the union of all green edges and E' is the union of both green and black edges.

V.II.IV Phase 4: Static assertion

As explained on section V.I.IV, our main objective in this phase is to have a transitive (black) edge of type **I** (instance) from the target node n_0 to source node n_2. That would mean, that we are looking for an **I** edge from type Car to the node *assertTy* (n_2), but that does not happen, since types **Car** and **House** are not directly related. If they were related, two other edges would be created connecting them. Those edges would be edges **Sb** and **Sp** and are explained in section V.III. If those edges existed, instance **I** would propagate from **House** to **Car** and then would be transitively propagated to node n_2, making it a valid instance of our problem. That is not the case, therefore, our framework would statically inform the user that **Car** is not one of the valid types for the object c.

V.III Branching and Inter-procedural Relationships

At the moment, our framework only checks for one objective at a time, and as mentioned on sections V.II.IV and V.I.IV, the main objective is to have an edge of type **I** (instance) from a target node to a source node. Thus, branching and inter-procedural analysis are not possible at the moment. Nevertheless, adding these would not change our overall layout, that happens because our grammar-guided graph is a much smaller version of the CFG that is output from Soot. That being said, we identified that for branching, we would have to check for two objectives instead of one. Let the consequent basic block of an *IfStmt* be the node n_0, the alternative basic block be n_1 and the source node (*assertTy*) be n_2. One objective would be having a transitive (black) edge of type **I** (instance) from the consequent node n_0 to source node n_2 and the other objective would be another transitive (black) edge of type **I** (instance) from the consequent node n_1 to source node n_2.

Likewise, inter-procedural analysis would be done by having one edge of a (probably) new type from one function to another function. We could then propagate the type of any object and keep the correctness of our approach. We know that this is possible since we keep our transivite relations throughout the program.

V.III Grammar definition

Wang et. al (2017) show that one of the most challenging parts of a grammar-guided reachability problem is to find and define a transitive grammar that is correct given a grammar-guided graph. Another challenge we face is that given a grammar-guided graph G and a grammar g and a pair of edges $e_1, e_2 \in E$, we can potentially have k edges from e_1 to e_2 and k edges from e_2 to e_1, where k is the total number of symbols in g, meaning that we might have a very large extended graph G', which could be $O(n^2 \cdot k)$. In spite of that, the Graspan project [5] demonstrates that scalability is not a problem when we use disk-base graph partitions. After some research, we discovered that the grammar below suffices in order to propagate a type of an object statically in our framework:

V.IV Grammar Labels

I = Instance: In our grammar, **I** edges are created whenever we have a *NewExpr* or *AssignExpr* that are not casts in our control flow graph.

Sb = Subtype: **Sb** edges are created from every subtype S to every supertype T. e.g. every type in the grammar-guided graph has a **Sb** edge to the **Object** class.

Sp = Supertype: **Sp** edges are created from every supertype S to every inherited type T. e.g the **Object** class has **Sp** edges to all types used in the grammar-guided graph.

W = Write: We create **W** edges whenever there is a write to a particular object.

R = Read: We create **R** edges whenever there is a read from a particular object.

C = TryCast: A **C** edge is created whenever we have a dynamic cast (*CastExpr*) in our control flow graph. This symbol is read "TryCast", as the casting operation is still going to be statically proven.

T = Type: A **T** edge is created whenever our control flow graph has a *NewExpr, AssignExpr* or a *Java Class* type.

V.IV.I Transitive production rules

As said before, it is important to notice that the transitive production rules of grammar g are created given g, a graph $G = (V, E)$, and all possible pairs of edges $e_1 = (u, v), e_2 = (v, w)$ in E, such that transitive edges are created from u to w if and only if the labels of edges e_1, e_2 are part of one of the transitive production rules in g. They are:

$$\begin{aligned} I &:= IW \\ &\mid IR \\ &\mid IT \\ &\mid SpI \end{aligned}$$

$$W := WT$$

$$\begin{aligned} T &:= TW \\ &\mid TR \\ &\mid SpT \\ &\mid SbT \end{aligned}$$

$$\begin{aligned} C := {} & CT \\ & \mid CR \\ & \mid CSb \\ & \mid IC \end{aligned}$$

A transitive **I** edge is created from u to w whenever e_1 has type **I** and e_2 has one of the types {**W**, **R**, **T**} or when e_1 has type **Sp** and e_2 has type **I**.

A transitive **W** edge is created from u to w whenever e_1 has type **W** and e_2 has type **T**.

A transitive **T** edge is created from u to w whenever e_1 has type **T** and e_2 has one of the types {**W**, **R**} or when e_2 has type **T** and e_1 has one of the types {**Sb**, **Sp**}.

A transitive **C** edge is created from u to w whenever e_1 has type **C** and e_2 has one of the types {**T**, **R**, **Sb**} or when e_1 has type **I** and e_2 has type **C**.

V.V Grammar-guided transitivity

In order to make our report complete, we are adding the algorithm for the grammar-guided reachability problem. The algorithm is a shorter version from the Graspan project. The algorithm is as following:

```
 Input: $G = (V, E)$, $g$
Output: $G' = (V, E')$
  for each $u \in V$ do
      $adj_u \leftarrow$ adjacency list of $u$
      while $adj_u \neq \emptyset$ do
          $v \leftarrow$ dequeue $adj_u$
          $adj_v \leftarrow$ adjacency list of $v$
          while $adj_v \neq \emptyset$ do
              $w \leftarrow$ dequeue $adj_v$
              if $(u, v)$ and $(v, w)$ is a rule $r$ in $g$ then
                  $E' \leftarrow E' \cup \{(u, w)$ of type $r\}$
                  $adj_u \leftarrow adj_u \cup \{w\}$
              end if
          end while
      end while
  end for
  return $G'$
```

VI. Expected Results

Considering the constraint of time, we could not get a detailed comparative study of our work with other related works. But we have a list of potential comparisons we can test against our work and also expect to see significant results.

The RTC tool [14] which utilizes several techniques by using static analysis to identify some cases in which instrumentation can safely be omitted for primitive types. It defined a flow-insensitive type-safety-level analysis, which classifies each lvalue expression in the program as safe, unsafe, or tracked, using a lattice. In spite of the performance improvements that RTC has achieved, it does not consider the interprocedural analysis of a program. However, our work is making use of Graspan [5] as the base model for deriving the transitive type relations. Graspan has validated the use of transitive reachability to perform interprocedural analysis. Keeping that into consideration we strongly believe that we would be able to extend the analysis to interprocedural in order to determine the type relations across the functions.

As the complexity of the program increases, the resulting grammar-guided graph's size increases considerably. In case of having a graph that does not fit into the memory, we would use the technique of partitioning the graph and process the derivation of transitive edges in parallel as motivated by Graspan [5]. In other words, whenever the size of a partition exceeds a threshold value, the graph is repartitioned so to fit in memory; the rest of the graph is kept in disk. Graspan supports both in-memory (for small programs) and out-of-core (for large programs) computation. Graspan is fully parallelized, allowing multiple transitive edges to be simultaneously added.

As mentioned in the related work section, Padhye and Sen et al. (2019) address the problem of dynamically checking if an instance of class S is also an instance of class T [16]. They propose fail-fast mechanism in this regard in order to minimize cache misses. Though their technique do not with multiple inheritance, separate compilation, and dynamic class loading, they proved that their fail-fasts implementation in the JVM provides 1.44×–2.74× speedup over HotSpot's native **instanceof**. We are confident that our work will eliminate even the minimal overhead of fast refutation of dynamic casting tests at runtime as all the dependency derivation would be done statically, as explained in previous sections.

VII. Conclusion and Future Work

We presented a scheme for performing static checking for dynamically-casted objects using transitive reachability over grammar-guided graphs. By evaluating the validity of dynamic casts at compile-time, we are confident that the overhead at runtime to determine the type of an object could be eliminated completely to yield better performance and reliable programs.

As our future, we intend to extend the algorithm implementation to determine the invalid type casts in interprocedural analysis as well as get it working for larger program graphs in order to compare and have concrete results against similar works to demonstrate the complete elimination of runtime overhead for dynamic casts. This shall in-turn make use of the graph partitioning technique [5] to achieve better performance against traditional graph processing.

The source code of the project can be found at https://github.com/luizperes/grammar-guided-static-type-checking.

References

[1] Docs.microsoft.com. (2019). Type conversions and type safety. [online] Available at: https://docs.microsoft.com/en-us/cpp/cpp/type-conversions-and-type-safety-modern-cpp?view=vs-2019 [Accessed 9 Dec. 2019].

[2] Docs.oracle.com. (2019). Chapter 5. Conversions and Promotions. [online] Available at: https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html#jls-5.5.1 [Accessed 9 Dec. 2019].

[3] Java rules for casting, J., Solanki, B., B., S. and Ramdhanie, V. (2019). Java rules for casting. [online] Available at: https://stackoverflow.com/questions/2233902/java-rules-for-casting/2233907#2233907 [Accessed 9 Dec. 2019].

[4] Examples Java Code Geeks. (2019). java.lang.ClassCastException - How to solve Class Cast Exception — Examples Java Code Geeks - 2019. [online] Available at: https://examples.javacodegeeks.com/java-basics/exceptions/java-lang-classcastexception-how-to-solve-class-cast-exception/ [Accessed 10 Dec. 2019].

[5] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, Ardalan Amiri Sani. Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code https://aftabhussain.github.io/documents/pubs/asplos17-graspan.pdf

[6] Arni Einarsson and Janus Dam Nielsen. A Survivor'sGuidetoJava Program Analysis with Soot http://cs.au.dk/ amoeller/mis/soot.pdf

[7] GitHub. (2019). Sable/soot. [online] Available at: https://github.com/Sable/soot [Accessed 11 Dec. 2019].

[8] Kai Wang and Guoqing Xu, University of California, Irvine; Zhendong Su, University of California, Davis; Yu David Liu, SUNY at Binghamton. GraphQ: Graph Query Processing with Abstraction Refinement—Scalable and Programmable Analytics over Very Large Graphs on a Single PC Usenix.org. (2019). [online] Available at: https://www.usenix.org/system/files/conference/atc15/atc15-paper-wang-kai.pdf [Accessed 11 Dec. 2019].

[9] Kai Wang, UCLA; Zhiqiang Zuo, Nanjing University; John Thorpe, UCLA; Tien Quang Nguyen, Facebook; Guoqing Harry Xu, UCLA. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine Usenix.org. (2019). [online] Available at: https://www.usenix.org/system/files/osdi18-wang.pdf [Accessed 11 Dec. 2019].

[10] GitHub. (2019). Graspan/graspan-cpp. [online] Available at: https://github.com/Graspan/graspan-cpp [Accessed 11 Dec. 2019].

[11] Erik Meijer and Peter Drayton - Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages Ics.uci.edu. (2019). [online] Available at: https://www.ics.uci.edu/ lopes/teaching/inf212W12/readings/rdl04meijer.pdf [Accessed 11 Dec. 2019].

[12] A Survivor's Guide to Java Program Analysis with Soot - Cs.au.dk. (2019). [online] Available at: http://cs.au.dk/ amoeller/mis/soot.pdf [Accessed 11 Dec. 2019].

[13] GitHub. (2019). Sable/soot. [online] Available at: https://github.com/Sable/soot [Accessed 11 Dec. 2019].

[14] Suan Hsi Yong and and Susan Horwitz - Using Static Analysis to Reduce Dynamic Analysis Overhead - Research.cs.wisc.edu. (2019). [online] Available at: https://research.cs.wisc.edu/wpis/papers/FMSD05.pdf [Accessed 11 Dec. 2019].

[15] Smlnj.org. (2019). ckit. [online] Available at: http://www.smlnj.org/doc/ckit/index.html [Accessed 11 Dec. 2019].

[16] Rohan Padhye, Koushik Sen - Efficient Fail-Fast Dynamic Subtype Checking - People.eecs.berkeley.edu. (2019). [online] Available at: https://people.eecs.berkeley.edu/ rohanpadhye/files/failfast-vmil19.pdf [Accessed 11 Dec. 2019].

[17] Sebastian Okon, Stefan Hanenberg - Can We Enforce a Benefit for Dynamically Typed Languages in Comparison to Statically Typed Ones? A Controlled Experiment - Ieeexplore.ieee.org. (2019). Can we enforce a benefit for dynamically typed languages in comparison to statically typed ones? A controlled experiment - IEEE Conference Publication. [online] Available at: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=7503719 [Accessed 11 Dec. 2019].

[18] Benjamin C. Pierce BC.Types and Programming Languages. The MIT Press: Cambridge, Massachusetts, 2002.