Pharo Smalltalk as a Basis For Development for the JVM

Dave Mason
Ryerson University, Canada
dmason@ryerson.ca

Abstract

Java, or more accurately the JVM, is ubiquitous - for the last 20 years it has ranked 1^{st} or 2^{nd} on the TIOBE list. Yet Java, the language, is also flawed - witness the dozens of languages that attempt to fix it while running on the JVM. Java is often used to support the thesis that explicit, or static, type systems are superior to latent, or dynamic, type systems. Despite this allegation, the evidence is that dynamic environments are significantly more productive environments in which to develop software, and Smalltalk provides the gold standard for IDEs.

This paper proposes marrying the two approaches: supporting software development in Smalltalk, while deploying to the ubiquitous JVM to inter-operate with the huge investment in libraries. While type inference - necessary to perform this magic - is a well-understood technology, we resolve some interesting challenges deriving from the complexities of the Java type system.

Keywords Language semantics, the JVM, Language interoperability, Type inference.

1. Motivation and State of Art

I volunteered to teach a 12-hour course to kids on "Programming Minecraft [19] Mods"[1]. There is a great book [10] to introduce kids to programming Minecraft in Java, but when I examined it, the first 50 pages were about setting up the environment and much more of the book was dedicated to dealing with the vagaries of Java. I thought, "There has to be a better way!" As a Smalltalk developer/researcher, I thought that Smalltalk would be that way.[2]

This paper proposes PharoJVM, which is an infrastructure (framework + tools) that allows developing and testing in Smalltalk for applications that ultimately run on the JVM. It is inspired by PharoJS [21] which allows development in Pharo Smalltalk and deployment to Javascript to run in a browser or `node.js`. The advantage is that the developer can work in a very fluid environment, with a best-in-class IDE, while deploying where needed.

1.1 Java and the JVM

Java and the JVM have been one of the dominant ecosystems for many years - ranking 1^{st} or 2^{nd} on the TIOBE index [29] consistently for almost two decades. This even before the Android platform adopted it as the basis for applications.

More important than Java is the JVM. In fact, James Gosling, primary creator of Java and the JVM has been quoted[3] as saying, "Most people talk about Java the language, and this may sound odd coming from me, but I could hardly care less. [the JVM is] what I really care about."

Despite the popularity of Java, there have been many criticisms of the language, but the importance of inter-operating with it on the JVM has led to the development of many other language systems designed and sold to develop and generate code to also work on the JVM. [4]

Improved Java Scala [26], Groovy [9], Kotlin [15], and xTend [33] are all Java-like languages that "fix" various aspects of the language. Scala uses type inference and uniform treatment of values to make a "purer" object-oriented language with functional programming features. Type inference is somewhat limited in that method parameters must be explicitly typed, and Scala types can sometimes be very complex. Groovy was originally a Java-flavored, dynamically-typed, scripting language for the Java programmer; as such it largely retains Java syntax - removing some boilerplate, and has more recently added static typing. Kotlin and xTend

[1] Minecraft is a very popular game with kids that runs on the JVM. Mods/extensions/plug-ins can be written to change behaviour.

[2] There are also books on programming Minecraft mods with Python [30, 25] that are quite good, but I didn't find them until later.

[3] `https://www.theserverside.com/news/2240037412/James-Gosling-Says-He-Doesnt-Care-About-Java`

[4] `https://en.wikipedia.org/wiki/List_of_JVM_languages`

are primarily syntactic sugar to clean up some of the annoyances of Java and are focussed on Java IDEs (IntelliJ and Eclipse, respectively).

Other Languages Versions of other languages have been created to target the JVM. Many are statically-typed, but this list focusses on dynamically-typed languages. Jython [14] implements Python. Clojure [3] is an updated Lisp that adds concurrency primitives. JRuby [12] implements Ruby. Redline Smalltalk [24] is a Smalltalk system that runs on the JVM. While any of these could have served for my original challenge, they were all file-based and lacked the IDE and simple integration that I was looking for. The Adventurous Developer's Guide to JVM Languages [16] has a good comparison of several of these languages.

1.2 Other approaches

The Truffle-Graal framework [6, 32] provides support for dynamic typing and might enable support for some of the more esoteric Smalltalk capabilities which we discuss in §5. When this project commenced we were unaware of this option, which seems worth exploring as there are currently Ruby and Javascript implementations using it. However, we remain interested in the performance possibility of compiling directly to statically-typed JVM code, and the possibility of other targets such as WebAssembly, and native machine code.

Strongtalk [2, 27] is a Smalltalk that generates high-performing static code where possible using a JIT compiler that uses speculative optimizations. Some of the development of Strongtalk led to the enhanced performance of the JIT on the JVM. We wanted to stay within Pharo, and our motivating example was to generate code to run on the JVM.

1.3 Contributions and outline

Our proposal aims at enabling developers to use a simpler language and a professional-grade IDE, and to build applications to run on the JVM platform. Currently we are more focussed on code written to run on the JVM, rather than taking generic Smalltalk code and producing JVM code for it. We have introduced PharoJVM: an infrastructure that allows programmers to use all the Pharo tools to build applications that are ultimately compiled to the JVM code. Our solution largely preserves Smalltalk semantics, while ensuring high performance and full interoperability.

In the remainder of this paper, we first discuss the issue of aligning the Smalltalk and Java type systems (§2). Then, §3 lists challenges arising from the differences between Smalltalk and Java, and how we have dealt with them. Next, we describe the structure of PharoJVM apps, the development process and how PharoJVM enables editting and testing Pharo code and deployment for the JVM (§4). §5 describes current limitations of the model. Lastly we conclude the paper and sketch some future work (§6).

2. Smalltalk versus the JVM Type Systems

Smalltalk is a dynamically-typed language, while the JVM is primarily statically-typed.[5] This means that to compile Smalltalk code to the JVM we need to find a single type for each identifier (local, instance, and class variable, parameter, class, and message) in the program. In a procedural or functional language, a sequence like: "`x:='abc'. x:=4`" will fail because there is no single type for `x`. However, this is rarely a problem, even in a procedural or functional language, because working programs must inherently have sensible type usage.

In an object-oriented language, "sensible type" means that the object understands the messages sent to it, so in the example, so long as the value stored in `x` - either a string or an integer - understands all the messages sent to it the code is fine. In a dynamically-typed language like Smalltalk, this is often called "duck typing" [18] where the determination takes place at run time; with the potential to cause a `doesNotUnderstand:` message to be sent if the current value does not undestand the particular message. In a statically-typed, object-oriented language, at first blush there is never a problem, because everything is an instance of "`Object`" so we can infer that `x` must be an `Object`. However we must ascertain, statically, that the message will always be understood, and `Object` does not understand all possible messages.[6] So in principle we must unify to the lowest common superclass of the objects we unify with the variable.

Similarly if we have a parameter to a method and that parameter is sent the message `abc`, we will not be able to call that method with two different objects, even if both understand the message `abc`, unless the invoked methods return compatible types.

Sometimes we want to have classes whose instances accept one or more messages, but which have no common superclass that accept all of the required messages, to be used in a similar way. In a dynamically-typed system there is no problem, because the classes we wish to use accept all of the methods. However, without some accommodation, there is no way to make this work for a statically-typed system. One solution, adopted by C++, is multiple inheritance - but that brings with it many problems. Another solution, adopted by Java, is class interfaces. An interface declare messages that the instances will understand (i.e. that have implementations of the corresponding methods).

Smalltalk and Java have similar class systems, with three notable differences:

1. Smalltalk classes are first-class objects, which will be discussed further in §3.3;

[5] the JVM now supports `invokedynamic` which allows some degree of flexibility, but dynamic typing is still complex to support.

[6] ...except in the sense that it will send `doesNotUnderstand` to most messages.

2. in Java a given method name for a class can be used for multiple methods, differentiated (overloaded) by the number and types of their parameters;
3. Java classes can implement one or more interfaces. An interface says that if a class implements the required instance methods, it can be treated as an instance of the interface and be statically checked as offering the required methods. As explained above, this is Java's attempt to emulate duck typing in a static type system. This could be emulated in Smalltalk with a trait that consisted solely of "`explicitRequirement`" methods - but there would be no point since there is no restriction on using any instance of any type, as long as all the methods actually invoked exist in the class.

The latter two differences will be discussed further in §3.5.

2.1 Type Inference

The Damas-Hindley-Milner type system [4] first came to prominence in the description of Standard ML [17]. It provides a linear-time algorithm for finding the most general type for lambda-calculus. It works by unification of parameters, function signatures, and results. It has been applied to the dynamically typed language Scheme in Soft typing [31].

Palsberg et al. [20] describe using type inference in object-oriented languages. Furr et al. [5] describe a static type inference system for Ruby. As they are addressing a similar language and goal (static typing), there are several similarities, such as Object types (objects which have not yet been associated with any class), but also several differences, such as union types which we handle differently to align with JVM interfaces.

Pluquet et al. [22] described a system that is available in Pharo as the Roel type checker which was oriented towards program understanding, were aiming for speed over precision of analysis. Suzuki [28] also looks at the application to Smalltalk.

2.2 Scalability and Application

Pluquet et al. [22] suggested that type inference does not scale well - primarily because they were looking to type all the classes and methods with no anchors to static types other than literals like numbers, characters, and booleans - so they were forced to sacrifice precision of analysis. However in our application, we only have to do type inference on the designated application class and all the classes to which it refers: directly or indirectly. Additionally we are generating code to inter-operate with Java libraries, and all of those libraries are statically typed, so Smalltalk identifiers are anchored to static types very quickly, so we contend that it will scale fine in our application. While we are doing whole-program type inference, the "whole program" is the particular application and the classes upon which it depends, rather than the whole Smalltalk image.

The type inference we require is different from Pluquet et al. because for us precision is paramount: if we don't have the exactly correct type, the resulting JVM code will not be accepted by the JVM loader.

The type inference is also complicated by interfaces. Interfaces are the JVM's way of mimicking duck-typing while avoiding multiple inheritance. Instances of two unrelated classes may be used in a particular context as long as they implement an appropriate interface - i.e. they implement a particular set of methods. These must be tracked so that the declaration of the type of a parameter or value can be either: the common superclass if it provides appropriate methods, or an appropriate interface that is declared to be provided by all of the classes.

3. Mapping Language Constructs

There are many subtle differences between Smalltalk and the JVM languages. However, for all the problems we have encountered so far, we have been able to find a functional alignment.

3.1 Blocks

Smalltalk BlockClosures conceptually are anonymous functions, though in fact they are instances with their own code associated with them. In JVM terms, they must each be implemented in a unique class.[7] They must also have access to free variables from their enclosing method - either read-only or read-write. This is very similar to the way BlockClosures are done in the Smalltalk VM - a heap-allocated object contains all of the variables that are accessed by both the method and the block. This allows BlockClosures to be first-class objects that can be passed or returned from a method.

```
1  eg1: param
2      | x y z |
3      x := 0.
4      z := y := 42.
5      #(2 5 7) asOrderedCollection
6          do: [: each | | w |
7              w := y + param;
8              x := x + w.
9              x > 10 ifTrue: [↑ y].
10     ].
11     ↑ x + z
```

Figure 1. Simple `BlockClosure` example

Figure 1 contains an example we will use to explain this.[8]

Figure 2 contains a simplified Java version of the JVM code that would be produced for the method in figure 1.

[7] Technically several blocks can share a single class if they have different numbers or types of parameters.

[8] Actually `do:` is handled specially, so the block wouldn't be created.

```
1  class resultInt extends Error {
2     int result;
3     resultInt(int result) {
4        this.result = result;
5     }
6  }
7  class freeVars {int x,y,param;
8     resultInt result;}
9  class blockClosure {
10    freeVars free;
11    void value(int each) {
12       int w;
13       w = free.y + free.param;
14       free.x = free.x + w;
15       if (free.x > 10)
16          throw(free.result=
17               new resultInt(free.y));
18    }
19 }
20 public class bcExample {
21    int eg1(int param) {
22       int z;
23       freeVars free = new freeVars();
24       free.param = param;
25       free.x=0;
26       z = free.y = 42;
27       try {
28          blockClosure bc =
29             new blockClosure();
30          bc.free = free;
31          OrderedCollection<Integer>temp=
32             new OrderedCollection<>();
33          temp.add(2);temp.add(5);
34          temp.add(7);
35          temp.do_(bc);
36       } catch (resultInt ri) {
37          if (ri==free.result)
38             return ri.result;
39          throw ri;
40       }
41       return free.x + z;
42    }
43 }
```

Figure 2. Java equivalent to simple `BlockClosure` example

There are many optimizations possible, but in the simplest case, if there are any free variables:

1. create a class to hold the shared variables (`freeVars`);
2. create a class for the block (`blockClosure`);
3. generate code to create and initialize an instance of the shared-variable class with the initial values (lines 23-26);
4. keep a reference to the instance in a local variable for the method (`free`);
5. generate an instance of the block class, initializing it with a reference to the shared-variable instance (`bc`, lines 28-30);
6. all references to the shared variables must be through the shared-variable (all the references to `free.` in both `eg1` and `value`).

All the blocks in a given method, whether nested or not, will normally use the same shared-variable object. In addition to working with Smalltalk code that calls `value`, `value:`, `value:value:`, et cetera, it also must implement the `java.util.function.Function` and `java.util.function.Consumer` interfaces so that it can work with the Java Collections Framework [11]. The `accept:` method simply calls `value:`. And, of course, these all have to have the correct types.

Because each block closure has its own environment and types, a unique class is created for each one.

Finally, when it comes to blocks for methods that are going to be inlined like `do:`, `ifTrue:ifFalse`, `whileTrue:`, and `timesRepeat` none of the above applies. They are simply handled as normal code generation.

In order for the idiom: "`aCollection do: #foo`" to work, symbols must respond to a `value` message of the correct arity, so `Symbol` becomes a subclass of `BlockClosure` on the JVM. Although `Symbol` is a subclass of `String` on Smalltalk, on the JVM `String` cannot be extended (subclassed) because it is declared as final. Because existing Smalltalk code expects that `'abc' = #abc` this will actually be translated into the equivalent of the Java `"abc".equals(new Symbol("abc").toString())`, which is tedious but easily handled as the types are all known.

3.2 Non-Local Exits

One of the most interesting features of BlockClosures are non-local exits. A return from a block causes a return from the method that created the block. This is directly implemented in the Smalltalk VM, but is not supported in the JVM, so must be emulated with exceptions. Figure 1 demonstrates this in lines 16,17 and 27,36-39.

The basic idea is to create an `Error` subclass for each type of value that needs to be returned. When a non-local

exit is executed in a `BlockClosure`, it creates an instance of that class, initializes its `result` value, and throws it.[9]

The code for the original method tries to catch the throw. Verifying that the caught object is the same as the `result` object allows this to work correctly even in the face of recursive methods. If it matches, then the proper `result` value is simply returned from this method. It it does not match, then it is re-thrown to be caught elsewhere.

3.3 Classes as objects versus `static`

Smalltalk classes are first-class objects, while the JVM classes are simply static code and have no inheritance structure. This means that class-side uses of `self` - other than calling methods - are trying to reference a non-existent class object, so are invalid. Class variables can be handled as static variables because there are no object boundaries that restrict access between classes and their instances. The inheritance structure is emulated by duplicating any referenced class-side instance variables and methods, as well as class variables, from superclasses. This maintains the semantics at the cost of some code duplication. Since there is no inheritance, any reference to `super` on the class-side may refer to an identically-named method, so name mangling may be required.

3.4 Class definition

We want people to be able to write normal, un-annotated, Smalltalk code and debug it on Pharo, and then to compile it for the JVM. For this to work well, the Smalltalk classes and methods that a Smalltalk developer is used to must translate to corresponding JVM classes and methods. There is a general mechanism, which is that Pharo classes that have a JVM equivalent class have a class-side method called `javaEquivalentClass` that returns the JavaClass to which it corresponds. As well every class may have a `javaMethodMap` which maps Smalltalk selectors to the JVM methods.

The `Collection` heirarchy is particularly tricky. Smalltalk `Array` maps to a special parameterized type with special code-generation to use the direct array accessing JVM instructions.

There are two kinds of classes that must be handled:

1. JVM classes which are extracted from `.class` or `.jar` files in the `CLASSPATH`[10] and have full static typing associated with them, including fields, methods, and interfaces.
2. Smalltalk classes that are being compiled to run on the JVM. Choosing the list of classes to be compiled starts with the application class and its superclasses. The class-side method `appClasses` provides additional classes that are associated with the application. This is extended by the transitive closure of classes that are referenced in the list of classes. The type of fields, methods, and interfaces must be derived because they are not extant in the dynamically-typed Smalltalk code.

   Symbols and block closures are actually special cases of this, with special code generation.

3.5 Message Sends

The most complex part of the type inference is message sends. It is significantly complicated by two properties of the JVM: methods are differentiated by name+type, and methods may be found either because they are in the class (or superclass), or because they are in an interface. We create a `JavaObjectType` object for each call site. There are three unifications required:

- When a message is sent to the object, the message type and result type must be added to the list of required message types. If there are already interfaces associated with the object, any interface that does not include a unifiable method is removed as inadequate for this use. If there are no interfaces, but there is already a class associated with the object, the message must be unifiable with at least one of the methods of the class.
- When a class is unified with the object (as a parameter to a method, or by assignment), the relationship to already unified classes, interfaces, and messages must be established. If this is the first class, it must support methods that unify with all of the required messages. The class becomes the class for the object, and its interfaces become the list of supported interfaces.

  If this is not the first class, then the class for the object must become the lowest common superclass of the existing class and the new class, and the interface list is intersected with the interfaces of the new class. All the required message types must be unifiable with one of the remaining interfaces, or with the class for the object.
- When an interface is unified with the object, all the message types of the interface are added to the required types list. All the required message types must be unifiable with one of the interfaces, or with the class for the object. The interface is added to the interface list. If there are any JVM classes that have been unified with this object, they must all implement the interface.

Once all unification has been complete (of all classes and all methods), all of the requirements need to be reviewed to unify each method in the method list.

If a Smalltalk class was included in the unification, and the object class resolves to an interface, then the JVM code generated must indicate that the class implements that interface.

[9] We actually create a unique `Error` subclass for each method that contains a block with a non-local return so that the JVM does the class match - for efficiency.

[10] `CLASSPATH` is the environment variable that tells the Java compiler and the JVM where to find library code.

Collections have a type variable for their element type, and any value added to the collection will be unified with the element type. In the code "`ar := #($A 3 'abc'), ar` will be an array whose element type has been unified with `Character`, `Number`, and `String` resulting in a type of `Object` (or, in general, any shared interface type), so only messages understood by `Object` (or any interface) will be legal.

3.6 `initialize` vs. constructors

The JVM allocates the space for an object and then calls a constructor. Smalltalk allocates the space for an object and then calls `initialize`. So `initialize` is equivalent to the parameterless constructor `<init>`.

Similarly, class-side `initialize` is similar to the class initializer, `<clinit>`.

3.7 Primitive Types

In Smalltalk everything is an object so messages can be sent to, e.g., integers. In the JVM the eight primitive types: byte, short, int, long, boolean, char, float, and double, are not objects, although there is a boxed (object) version of each. `Integer`, the boxed version of `int`, is a first-class object, but it pays a significant cost, being heap allocated and garbage collected, and an indirect reference being required to access the internal `int` value. Most of the methods of e.g. `Integer` are available as static methods with a parameter of `int`. So if `i` is an `Integer`, one can ask for `i.toString()` but the static method `Integer.toString(42)` also works. We keep things as primitive types wherever possible, including using static methods where possible, and only create the boxed version where it has to unify with an `Object`, such as in collections. In the JVM all collection types except arrays must be `Objects` and arrays do not understand any messages. Therefore arrays operations must be recognized and converted into regular code or call static methods in `java.util.Arrays` to manipulate them.

4. App Development with PharoJVM

An application to be compiled by PharoJVM is simply represented as a class that is a subclass of JavaExecutable. The message "`MyJVMApplication classFile`" will generate all the necessary JVM `.class` files. A `.class` file will be produced for the application class and all referenced classes and their superclasses, as well as inner-class files for `BlockClosures` and shared variables. Each `.class` file will contain only the methods that are potentially referenced.

`JavaExecutable` adds `JavaGlobals` as a poolDictionary as a convenience, and has a few class-side methods to output the required `.class` or `.jar` files. JavaGlobals is populated automatically with package paths for all the paths defined in the loaded `CLASSPATH` class and jar files. This allows expressions like `java util Vector` to translate to the appropriate Java class.

Currently we are more focussed on code written to run on the JVM, rather than taking generic Smalltalk code and producing JVM code for it.

Example

Just for demonstration purpose, we present a trivial example program in figure 3. It was a class-side method for an example class. Any class-side method called `main:` with a single parameter is unified with a method type with an array of strings as the parameter and no returned value - the Java type `([Ljava/lang/String;)V`. So the `ignored` parameter is unified against an array of Strings - which, since it's unused, isn't a conflict. The result is unified with `void` - which since there is no explicit return, also isn't a conflict. `ar` is unified with an array of `int` of size 20, `tot` is unified with `int`, `flag` is unified with `boolean`, and `each` is unified with the element type of `ar`.

```
1  main: ignored
2      | ar tot flag |
3      ar := #(1 2 3 4 5 6 7 8 9 10
4          11 12 13 14 15 16 17 18 19 20).
5      tot := 0.
6      flag := true.
7      10000000 timesRepeat: [
8          ar do: [: each |
9              flag ifTrue: [
10                 tot := tot + each
11             ] ifFalse: [
12                 tot := tot - each
13         ]].
14         flag := flag not.
15     ].
```

Figure 3. A trivial main program

Figure 4 shows the raw execution times for the example program run in Pharo and the JVM.[11] [12] Pharo inlined is where the `timesRepeat:` and `do:` have been inlined into loops using `whileTrue:`. Execution time for an equivalent C program are shown purely for demonstration purposes. Although speed wasn't the motivation for this research, this shows that a significant speed-up (about $5\times$ to $7\times$) is available, with an additional $10\times$ in C.

5. Limitations

Currently we are more focussed on code written to run on the JVM, rather than taking generic Smalltalk code and producing JVM code for it. Also, the program in figure 3 is

[11] Java HotSpot(TM) 64-Bit Server VM (build 25.112-b16, mixed mode)

[12] 2.9 GHz Intel Core i5

Pharo 6	3.083 sec
Pharo 6 inlined	1.991 sec
the JVM	0.476 sec
cc -O2	0.047 sec

Figure 4. Execution times for the trivial program

about the limit of what we can currently compile. Therefore, there may be complexities in supporting some system Pharo classes and methods on the JVM.

This project is currently in active development, so the following current limitations are believed to be resolvable:

1. `perform:` could be handled with the JVM's reflection mechanism.
2. There is no bridge between the Pharo image and the running JVM. A limited capability to debug in Pharo while accessing native JVM libraries - similar to PharoJS- could be developed.
3. Continuations could be supported using Java continuations libraries (for example, Quasar [23]), which could enable coding of Seaside applications.

The following are more fundamental limitations for which we currently see no resolution:

1. `thisContext` would require access to the stack frame, and there is currently no JVM API available.
2. `become:` requires functionality not currently available in the JVM.
3. Exceptions are non-resumable.
4. `adoptInstance:` and related class-changing would require JVM API which is not currently available (because of interaction with JITs - among other reasons).
5. Classes in Smalltalk are first-class objects, whereas in the JVM they are static, non-objects, so classes cannot be assigned to variables, passed as parameters, or returned as results, except possibly if the use is monomorphic.

6. Conclusion and Future Work

This paper describes preliminary work. We are currently at the stage of compiling toy applications, so more challenges will doubtless present themselves as we scale up, and as we gain more experience we'll extend in useful ways. Particularly, we would like to add features so that more debugging can be done within Pharo by providing mocks for things like System and I/O and/or using a bridge and proxies similar to that done in PharoJS. Ideally many applications could be developed completely within Pharo, and then exported to the JVM. Conversely, it would be nice to be able to take existing Smalltalk applications and export them to run on the JVM.

BlockClosures are currently compiled to separate JVM Inner Classes [13]. Java lambdas are handled as methods for single-method interfaces that generate an inner class using `invokedynamic` rather than with an external `.class` file. Although there does not appear to be a significant performance difference [13]and there appear to be limitations, we would like to explore this as an alternative.

Currently, if a compilation unit can't be completely statically typed, it fails. Gradual Typing [1, 7, 8] would allow everything to work until an un-statically-typed value was encountered at run time, at which time an exception would be thrown.

The static typing derived from the analysis described here would also be applicable to a native compiler, and as figure 4 shows there is significant performance to be obtained. The same would apply to WebAssembly which would allow targeting web applications. PharoJS could benefit from the type analysis described here to generate much better Javascript code where a concrete type can be determined - falling back to the existing generic code where appropriate.

It would be interesting to compare with Truffle/Graal.

Acknowledgments

Many thanks to the anonymous reviewers for many excellent comments that have significantly improved the paper, as well as several challenges to our thinking regarding the implementation.

References

[1] Esteban Allende et al. "Gradual Typing for Smalltalk". In: Sci. Comput. Program. 96.P1 (Dec. 2014), pp. 52–69. ISSN: 0167-6423. DOI: 10.1016/j.scico.2013.06.006. URL: http://dx.doi.org/10.1016/j.scico.2013.06.006.

[2] Gilad Bracha and David Griswold. "Strongtalk: Type-checking Smalltalk in a Production Environment". In: Proceedings of the Eighth Annual Conference on Object-oriented Pr OOPSLA '93. Washington, D.C., USA: ACM, 1993, pp. 215–230. ISBN: 0-89791-587-9. DOI: 10.1145/165854.165893. URL: http://doi.acm.org/10.1145/165854.165893.

[3] Clojure. URL: https://clojure.org (visited on 2018-08-15).

[4] Luis Damas and Robin Milner. "Principal Type-schemes for Functional Programs". In: Proceedings of the 9th ACM SIGPLAN POPL '82. Albuquerque, New Mexico: ACM, 1982, pp. 207–212. ISBN: 0-89791-065-6. DOI: 10.1145/582153.582176. URL: http://doi.acm.org/10.1145/582153.582176.

[5] Michael Furr et al. "Static Type Inference for Ruby". In: Proceedings of the 2009 ACM Symposium on Applied Computin SAC '09. Honolulu, Hawaii: ACM, 2009, pp. 1859–1866. ISBN: 978-1-60558-166-8. DOI: 10.1145/

[13] see https://stackoverflow.com/questions/19001241/big-execution-time-difference-between-java-lambda-vs-anonymous-class

1529282.1529700. URL: http://doi.acm.org/10.1145/1529282.1529700.

[6] GraalVM Project. URL: https://www.graalvm.org (visited on 2018-08-01).

[7] Gradual Typing. URL: https://en.wikipedia.org/wiki/Gradual_typing (visited on 2018-06-13).

[8] Gradualtalk. URL: https://pleiad.cl/research/software/gradualtalk (visited on 2018-06-15).

[9] Groovy. URL: http://groovy-lang.org (visited on 2018-08-15).

[10] Andy Hunt. Learn to Program with Minecraft Plugins. The Pragmatic Bookshelf, 2014. ISBN: 978-1-937785-78-9.

[11] Java Collections Framework. URL: https://docs.oracle.com/javase/9/docs/api/java/util/package-summary.html (visited on 2018-08-19).

[12] Jruby. URL: http://jruby.org (visited on 2018-08-15).

[13] JVM Inner Class. (Visited on 2018-08-01).

[14] Jython. URL: http://www.jython.org (visited on 2018-08-15).

[15] Kotlin. URL: https://kotlinlang.org (visited on 2018-08-15).

[16] Rebel Labs. The Adventurous Developer's Guide to JVM Languages. URL: https://zeroturnaround.com/rebellabs/the-adventurous-developers-guide-to-jvm-languages-java-scala-groovy-fantom-clojure-ceylon-kotlin-xtend/ (visited on 2018-06-13).

[17] Robin Milner. "A Proposal for Standard ML". In: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming. LFP '84. Austin, Texas, USA: ACM, 1984, pp. 184–197. ISBN: 0-89791-142-3. DOI: 10.1145/800055.802035. URL: http://doi.acm.org/10.1145/800055.802035.

[18] N. Milojkovic, M. Ghafari, and O. Nierstrasz. "It's Duck (Typing) Season!" In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC). May 2017, pp. 312–315. DOI: 10.1109/ICPC.2017.10.

[19] Minecraft. URL: https://minecraft.net/ (visited on 2018-06-14).

[20] Jens Palsberg and Michael I. Schwartzbach. "Object-oriented Type Inference". In: Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications. OOPSLA '91. Phoenix, Arizona, USA: ACM, 1991, pp. 146–161. ISBN: 0-201-55417-8. DOI: 10.1145/117954.117965. URL: http://doi.acm.org/10.1145/117954.117965.

[21] Pharo JS. Develop in Pharo - Run on Javascript. URL: http://pharojs.org (visited on 2018-06-13).

[22] Frédéric Pluquet, Antoine Marot, and Roel Wuyts. "Fast Type Reconstruction for Dynamically Typed Programming Languages". In: Proceedings of the 5th Symposium on DLS '09. Orlando, Florida, USA: ACM, 2009, pp. 69–78. ISBN: 978-1-60558-769-1. DOI: 10.1145/1640134.1640145. URL: http://doi.acm.org/10.1145/1640134.1640145.

[23] Quasar. URL: http://www.paralleluniverse.co/quasar/ (visited on 2018-08-01).

[24] Redline. URL: http://www.redline.st (visited on 2018-08-15).

[25] Craig Richardson. Learn to Program with Minecraft - Transform You No Starch Press, 2015. ISBN: 978-1-59327-670-6.

[26] Scala. URL: https://www.scala-lang.org (visited on 2018-08-15).

[27] Strongtalk. URL: http://www.strongtalk.org/ (visited on 2018-08-19).

[28] Norihisa Suzuki. "Inferring Types in Smalltalk". In: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Pr POPL '81. Williamsburg, Virginia: ACM, 1981, pp. 187–199. ISBN: 0-89791-029-X. DOI: 10.1145/567532.567553. URL: http://doi.acm.org/10.1145/567532.567553.

[29] TIOBE. Language Rankings. URL: https://www.tiobe.com/tiobe-index/ (visited on 2018-06-13).

[30] David Whale and Martin O'Hanlon. Adventures in Minecraft, 2nd Ed John Wiley & Sons, 2017. ISBN: 978-1-119-43958-5.

[31] Andrew K. Wright and Robert Cartwright. "A Practical Soft Type System for Scheme". In: ACM Trans. Program. Lang. S 19.1 (Jan. 1997), pp. 87–152. ISSN: 0164-0925. DOI: 10.1145/239912.239917. URL: http://doi.acm.org/10.1145/239912.239917.

[32] Thomas Würthinger et al. "One VM to Rule Them All". In: Proceedings of the 2013 ACM International Symposium on Onward! 2013. Indianapolis, Indiana, USA: ACM, 2013, pp. 187–204. ISBN: 978-1-4503-2472-4. DOI: 10.1145/2509578.2509581. URL: http://doi.acm.org/10.1145/2509578.2509581.

[33] Xtend. URL: https://www.eclipse.org/xtend/ (visited on 2018-08-15).