Duck Futures: A Generative Approach to Transparent Futures

Eric Lin Ganesha Upadhyaya Sean Mooney Hridesh Rajan
Iowa State University, Ames, IA, 50011, USA
{eylin, ganeshau, smooney, hridesh}@iastate.edu

Abstract

Futures offer a convenient abstraction for encapsulating delayed computation. It is a mechanism to introduce concurrency through a rewrite of the sequential program. However, managing futures is tedious and requires knowledge of concurrency and its concerns. The notion of transparent futures is used to hide the complexity of futures from developers. A number of techniques based on transparency have been proposed to create and manage futures. Previous techniques make use of reflection. In this paper, we propose *duck futures* that use a generative approach. We show that duck futures are much more efficient compared to previous notions of transparent futures. We also present the first large scale study of the applicability and utility of duck futures in practice using the Boa infrastructure for mining large scale open source repositories. Our study finds that transparent futures, despite their limitations, can be very useful in practice.

1. Introduction

Futures act as a placeholder for the result of the not-yet-performed method invocation. When a client makes an asynchronous call, a future object is returned immediately and lets the client to proceed with its computation. Later when the client uses this future, it is blocked until the result is available. In a way, futures introduce concurrency to the sequential program with small rewrite.

Many concurrency mechanisms such as active objects, actors, separates, active containers, asynchronous references and many more use futures. Halstead in MultiLisp [15] popularized the notion of futures. Futures are applicable to statically-typed, object-oriented languages [2, 18, 22, 23]. Languages such as Eiffel [9] and Io [1] use active objects [16] which return futures. Despite the popularity, creating and managing futures can be tedious. It requires refactoring of the code and inter-procedural program analysis to track if any futures are passed across method boundaries.

The complexity of creating and managing futures is hidden from developers using the notion of transparency. Fully-transparent futures [3, 8], semi-transparent futures [24], futures using proxies [19] etc, provides transparent futures to developers. In fully-transparent futures, asynchronous behavior is entirely masked from the developer. ProActive [8] and Mandala [3] are two approaches that provides fully-transparent futures. ProActive uses inheritance for this purpose. A proxy class that extends the return type is created dynamically during method calls using reflection, and then returned to act as a future. Similar to ProActive, Mandala uses reflection to create futures. The main difference being Mandala imposes constraints that the return type has to be an interface which limits its applicability, but eliminates other inheritance issues. Pratikakis *et al.* [19] also proposed using proxies as futures to achieve transparency.

Previous approaches make use of reflection to obtain a proxy class of the returned type. However, reflection doesn't come for free. Obtaining a class's metadata has a substantial cost, and using code produced by reflection is more expensive because of poorer code optimization. The overhead can overshadow implicit concurrency benefits of futures. We believe that reflection is not necessary for creating futures. The type information necessary to create futures can be obtained statically, and all the runtime cost using reflection can be moved to compile time, which can effectively increase runtime performance. In this paper, we propose *duck futures* to create transparent futures. Unlike previous approaches, duck futures are generated statically at compile time as necessary, i.e. futures are generated only for classes that are being used as return types. Our static approach allows us to avoid class reflection completely. Like previous approaches, duck futures have the same type as the original result object. Hence, it can be used as the value at all the places where the original result object is expected and no refactoring of the client code is necessary. Like other transparent futures, duck futures are implemented using wait-by-necessity mechanism [7]. Duck futures are successfully implemented as part of the Panini [4, 6, 20, 21] compiler.

[Copyright notice will appear here once 'preprint' option is removed.]

Java program without Futures

```
1  class ComplexT {
2    ComplexT getFirstPart() { ... }
3    ComplexT getSecondPart() { ... }
4  }

6  class Factory {
7    ComplexT create(ComplexT c1, ComplexT c2) { ... }
8    ComplexT create(...) { ... }
9  }

11 class Client {
12   Factory f = new Factory();
13   ComplexT work () {
14     ComplexT c1 = make1();
15     ComplexT c2 = make2();
16     return f.create(c1, c2);
17   }
18   private ComplexT make1() {
19     // ...
20     ComplexT first = f.create (...) ;
21     return first ;
22   }
23   private ComplexT make2() {
24     // ...
25     ComplexT second = f.create(...);
26     return second;
27   }
28 }
```

Java program with Futures

```
1  class ComplexT {
2    ComplexT getFirstPart() { ... }
3    ComplexT getSecondPart() { ... }
4  }
5  class Factory {
29   /* ComplexT */ Object create(ComplexT c1, ComplexT c2) { ... }
30   /* ComplexT */ Object create(...) { ... }
31 }
32 class Client {
33   Factory f = new Factory();
34   ComplexT work () {
35     ComplexT c1 = make1();
36     ComplexT c2 = make2();
37     return f.create(c1, c2);
38   }
39   private ComplexT make1() {
40     // ...
41     Object futureFirst = f.create (...) ;
42     if ( futureFirst instanceof Future)
43       futureFirst = futureFirst .get() ;
44     ComplexT first = (ComplexT) futureFirst;
45     return first ;
46   }
47   private ComplexT make2() {
48     // ...
49     Object futureSecond = f.create (...) ;
50     if (futureSecond instanceof Future)
51       futureSecond = futureSecond.get();
52     ComplexT second = (ComplexT) futureSecond;
53     return second;
54   }
55 }
```

Figure 1. A program to illustrate that managing futures is tedious. The left-hand-side contains a Java program and the right-hand-side shows the refactored program using java.util.concurrent.Future. The highlighted lines indicates the refactored code.

We first describe our work on duck futures in §3. Then in §4, we evaluate the performance overhead of duck futures and compare it against ProActive and Mandala. There are many previous work on transparent futures, but no large scale empirical study has been conducted on how applicable and useful they are in practice. Thus, we investigate the applicability and utility of transparent futures in practice by performing a large scale open source repository mining using Boa [10]. The contributions of this work and some interesting results we have discovered include:

- We show that generating inheritance based transparent futures can be done statically, removing the cost of future creation from runtime completely. This could potentially make transparent futures more attractive feature for implicitly concurrent languages.
- We show that duck futures has a significantly lower performance overhead compared to ProActive and Mandala.
- Duck futures, like transparent futures provide implicit concurrency, and maximize benefiting from concurrency when the future is not being used immediately. Our study on the distance of method calls and use of the return value in practice shows that return values are usually used at a few statements away from the call, thus allowing the program to benefit from concurrency.
- In practice, a fair amount of futures remain unclaimed within the same method and may potentially cross the boundary of methods. Without transparency, interprocedural refactoring is needed for these cases.
- Classes that are incompatible with transparent futures are not very commonly written by programmers. In addition, final classes are rarely used as return types. In contrast, methods returning primitive types are fairly common.

2. Motivation

Managing futures is tedious for a number of reasons. First, there is a need to refactor client code. This refactoring involves carefully replacing the return value of a method call with a future so that the method call can be made asynchronously. Second, if the returned value crosses method boundary then either futures needs to be claimed, or an interprocedural analysis followed by instrumentation needs to be performed. Figure 1 shows an example that illustrate both problems.

The example Java program shown in Figure 1 contains three classes, ComplexT, Factory and Client. The left-hand-side highlights parts of the code that requires refactoring to use futures. The right-hand-side highlights refactored code using java. util .concurrent.Future. The methods make1() and make2() of Client receives an object from Factory (line 41 and line 49 respectively). If the received object is an instance of

Future then retrieves the actual value of the future using get() method on Future object. In both cases, actual object (type ComplexT) is returned. Lines 43 and 44 is used to claim the future, as futures must be claimed within the extent of the method. To determine proper method boundaries of futures, an interprocedural program analysis is required. Hence, to use futures in the program a substantial refactoring of the program is required. And, to determine when and where all the future objects needs to be claimed requires an interprocedural program analysis.

3. Duck Futures: Design and Implementation

In this section we describe the design and informal semantics of duck futures. Basic idea is that a duck future is an object that behaves exactly like the original result object (walks like a duck and swims like a duck and quacks like a duck it is a duck). In other words, it has the same type as the original result object. Therefore, all of the methods that can be invoked on the result object can be invoked on the duck future. These invocations are delegated to the original result object when it is available.

Since the duck future has the same type as the original result object, it can be used at all the places where the original result object is expected. Therefore no refactoring of the client code is necessary as in the right-hand-side of Figure 1.

We have added support for duck futures in the Panini compiler which extends the OpenJDK java compiler. To implement this semantics, futures are hidden behind a carefully constructed interface that is automatically generated by the compiler. We first briefly explain the compilation strategy and then demonstrate it using an example.

Essentially, if a class C is to be wrapped in a duck future, we extract C's public signature as an interface. We then generate an implementation of that interface as a Proxy [7]. This implementation takes care of managing the future, the claimed value, and the appropriate delegation to the claimed value.

3.1 Hiding behind an interface

Interfaces provide languages like Java a type-safe mechanism to abstract what methods may be called on a type from the implementation of each method. The interface mechanism can be used to hide the details of managing dispatching and claiming a future from the client. Figure 2 shows Duck Future interface. Every class that needs to be wrapped in an duck future will implement this interface. For the example program shown in Figure 1, ComplexT class is used as return value by Factory and Client classes. Hence, ComplexT needs to be wrapped in a duck future as shown in Figure 3.

3.2 Inside a Duck Future

Figure 3 shows the duck future generated for ComplexT class. DuckFuture_ComplexT class needs to extend ComplexT and

```
1  public interface DuckFuture<T> {
2    // Method to claim the actual object
3    public void finishFuture(T t);
4    // Returns the actual object wrapped inside the duck future
5    public T get();
6  }
```

Figure 2. Interface of a duck future.

```
1  class DuckFuture_ComplexT extends ComplexT implements DuckFuture
       {
2    // Wrapped actual object
3    private ComplexT wrapped = null;
4    // boolean to indicate if the future has been claimed or not
5    private boolean isRedeemed = false;
6    // Translated method
7    public final ComplexT getFirstPart() {
8      if (isRedeemed == false) this.get();
9      return wrapped.getFirstPart();
10   }
11   // Translated method
12   public final ComplexT getSecondPart() {
13     if (isRedeemed == false) this.get();
14     return wrapped.getSecondPart();
15   }
16   // Method to fill the wrapped object with the actual object
17   public final void finish (ComplexT t) {
18     synchronized (this) {
19       wrapped = t; isRedeemed = true;
20        notifyAll () ;
21     }
22   }
23   // Claim method
24   public final ComplexT get () {
25     // Block waits until the actual object has been claimed
26     while (isRedeemed == false) try {
27       synchronized (this) {
28         while (isRedeemed == false) wait();
29       }
30     } catch (InterruptedException e) {
31     }
32     return wrapped;
33   }
34 }
```

Figure 3. Generated DuckFuture for ComplexT class shown in Figure 1

```
1  class Factory {
2    public ComplexT create(ComplexT c1, ComplexT c2) {
3      final DuckFuture duckFuture = new DuckFuture_ComplexT();
4      new Thread(new Runnable(){
5        void run() {
6          // Original body of create, creates ComplexT c
7          duckFuture.finish(c);
8        }
9      }).start();
10     return duckFuture;
11   }
12 }
```

Figure 4. Factory class with public method create refactored .

implement DuckFuture interface. DuckFuture_ComplexT contains reference to Future object (wrapped) on line 3 and on line 5 a boolean (isRedeemed) which indicates if the future has been claimed or not. lines 7-10 shows the translated getFirstPart() method of ComplexT class. Each such method

```java
1  class Client {
2    Factory f = new Factory();
3    public final ComplexT work() {
4      DuckFuture duckFuture = new DuckFuture_ComplexT();
5      new Thread(new Runnable(){
6        void run() {
7          ComplexT c1 = make1();
8          ComplexT c2 = make2();
9          ComplexT c = f.create(c1, c2);
10         duckFuture.finish(c);
11       }
12     }).start();
13     return duckFuture;
14   }
15   private ComplexT make1() {
16     // ...
17     ComplexT first = f.create (...) ;
18     return first ;
19   }
20   private ComplexT make2() {
21     // ...
22     ComplexT second = f.create(...);
23     return second;
24   }
25 }
```

Figure 5. Client class with public methods refactored.

is implemented by checking if the actual value is available (isRedeemed), calling the claim method (get() from lines 24-33) if it is not (which will block until the value of the future is available), and finally delegating the call to the actual value object (wrapped). All of the methods ComplexT and ComplexT inherited from its superclass will need a translated method inside the DuckFuture. The translation strategy only needs the signature information of the methods, which can be obtained while compiled statically.

3.3 Automatically claiming a future

In the previous section we wrapped a ComplexT class using DuckFuture. To illustrate the usage of DuckFuture consider an example concurrent implementation of Factory and Client classes. In Figure 4, the public method create (...) in the Factory class is refactored to make use of DuckFuture. In the modified method, a duck future of the type of DuckFuture_ComplexT is created, and immediately returned. On lines 4-9, a separate thread is created and started, and a ComplexT instance is created using the original code. The ComplexT object is then delegated to the DuckFuture. The same refactoring process is applied to all the public methods in the Client class. The refactored Client class is shown in Figure 5.

To understand how a future is claimed automatically, consider the lines 7-9 of Figure 5. On line 7 a create (...) call is made to Factory. The create (...) method of Factory returns a DuckFuture c1 right away. Client can now proceed to line 8 for a second create (...) call to Factory and receive another Duck Future c2. Finally, on line 9 a create(c1,c2) call is made to Factory. At this time, the value of the values of both c1 and c2 have to be claimed to be able to make a successful call. By this time if the previous calls to the Factory class has been completed, Client will then receive the actual values of c1 and c2. Otherwise, Client blocks.

On the other side, Factory received two create (...) calls. Factory processes them in the order they are received and calls the finish () method on the Duck Future created with the returned value. The finish () method then puts the actual return object and notifies to any consumer who is blocked. In our case, Client is notified that, Factory has finished computing and c1 and c2 are both available. Now, Client can proceed and make the call in line 10.

3.4 Benefits

This technique allows clients to remain completely oblivious to the details of managing a future. It is ensured the mechanics are always implemented correctly because the translation is machine generated. Another benefit is that Futures can be used without any noticeable impact at client sites, for asynchronous computation. Finally, all the work is done at compile time, and adds little overhead to runtime.

4. Evaluation

In this section we describe the research questions we wish to answer (§4.1), we show the experiments we have performed to answer these research questions (§4.2), and the results of the experiments (§4.3).

4.1 Research Questions

RQ1: *How much overhead is added when constructing a transparent future?* Futures are used to allow asynchronous calls and introduce concurrency to programs. To create these futures an overhead is inevitably added to the asynchronous method invocation process compared to synchronous method calls. We want to know how much overhead is added by each of the approaches presented in this paper.

RQ2: *How much claiming overhead is added when attempting to use a future?* All of the approaches mentioned in this paper use a wait-by-necessity mechanism such that attempting to claim a future blocks if result is not available. However, checking the completion state of the future and redirecting the method calls to the actual object adds an extra overhead that using a normal object doesn't have. This is another performance overhead introduced by futures. We would like to know how duck futures perform compare to other works w.r.t. this overhead.

RQ3: *Does duck futures's generative approach gives us a performance advantage compared to reflective approaches?* Duck futures uses a generative approach, which statically generates the futures at compile time. By contrast, other transparent future approaches create a reflections of the future object dynamically. We wish to understand the advantages of avoiding the use of reflection.

RQ4: *What is the potential implicit concurrency we can get out of transparent futures?* Using futures adds an overhead to a method call. In exchange, concurrency is gained by allowing the client to proceed with its computation while the method call is executed asynchronously. The benefit is maximized if the method calls are made as soon as possible, and the future is claimed as late as possible. However, when using transparent futures, programmers do not necessarily write their programs with concurrency in mind. We would like to know in practice, what the call-claim distance normally is, i.e. the distance between a method call and using the result of the method call in terms of number of statements?

RQ5: *How applicable are transparent futures in practice?* Futures using an inheritance approach generate a future subtyping the target class. Final classes and primitive types cannot be properly subtyped. Even with non final classes, those with final methods or public fields can also be an issue because the wait-by-necessity mechanism cannot be applied. More discussion on completeness of duck futures and transparent futures as a whole is described in §5.1. We would like to know about the importance of this issue in practice.

4.2 Approach of Evaluation

In this section, we describe our evaluation approach to answer the research questions identified in §4.1.

4.2.1 RQ1, 2 and 3: Evaluation Setup and Approach

To measure the performance overhead, we designed two microbenchmarks to specifically measure the two use cases of using a future: the creation of the future and returning a future to the caller, and using the future by invoking a method call on the future object.

We use a measurement methodology taken from Georges *et al.* [14]. To measure the runtime performance of these benchmarks, we warm up the JVM until steady-state is reached, by measuring 3 consecutive runs with a coefficient of variation under 0.02. We then obtain the steady-state performance runtime, by measuring the mean of ten steady-state runs.

The experiments were run on a single machine with 24 GB of memory and 24 cores clocked at 400 MHz, running on Linux 3.5.6-1.fc17 kernel with OpenJDK 64-Bit Server VM build 23.2-b09. The Panini compiler version 0.9.3 [4] was used for the Panini programs, Mandala version 2.3 [3] and ProActive Programming 5.4.2 [8] were used for Mandala and ProActive respectively.

4.2.2 RQ4 and 5: Evaluation Approach

For research questions 4 and 5, we look into source code of open source repositories to find answers to the questions. To do a large scale analysis on a large amount of source code, we make use of Boa [10] and its dataset from SourceForge [5]. Boa is an infrastructure for mining software repositories at a large scale. It allows users to write their own queries and extract data of source code and project details they are interested in from repositories. The dataset we used for our evaluation is the full September 2013 dataset of SourceForge provided by Boa. This dataset contains 35,341 Java projects including widely-used Java projects, such as Azureus/Vuze, Weka, Hibernate, JHotDraw, JabRef, JUnit, iText, FindBugs, JML, TightVNC, etc. and has been used for several studies e.g. [11, 12].

4.3 Experiment Results

4.3.1 Construction Overhead of Transparent Futures

To measure the overhead of the transparent future implementation, we constructed a microbenchmark that focuses on the invocation of a method and obtaining a future. The program repeatedly sends asynchronous calls to another object, obtaining a future object and claiming it for 1,000,000 number of times. The method being called has the minimal amount of execution and only contains a single return statement.

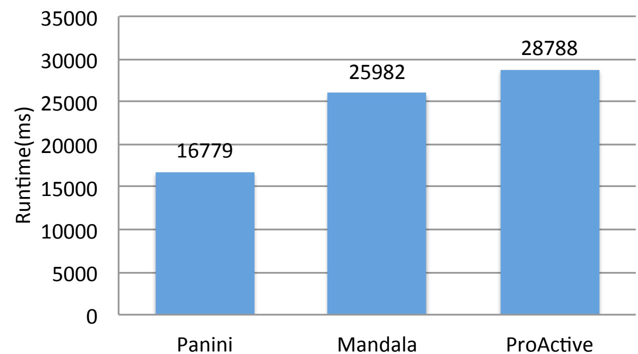

Figure 6. Runtime performance overhead of 1,000,000 asynchronous calls.

The results shown in Figure 6 represents the mean of 10 steady state runtime in milliseconds. At 1,000,000 iterations, duck futures has a 54.85% performance gain over Mandala and 71.57% over ProActive.

The results shows that the use of reflection to generate proxies impacts the performance of method invocations greatly.

4.3.2 Claim Overhead of Transparent Futures

In this section, we measure the overhead of claiming a future by using another microbenchmark. The benchmark is designed to test the overhead of the actions of checking the completion state of the future, and method invocation on the actual result. The program first makes an asynchronous call and then immediately claims the future. The claim will be blocked until the method is done executing and the future is completed. The program then attempts to invoke a method on the same completed future for another 10,000,000 times.

To act as a base case for comparison, another program was tested where the receiver is a normal object.

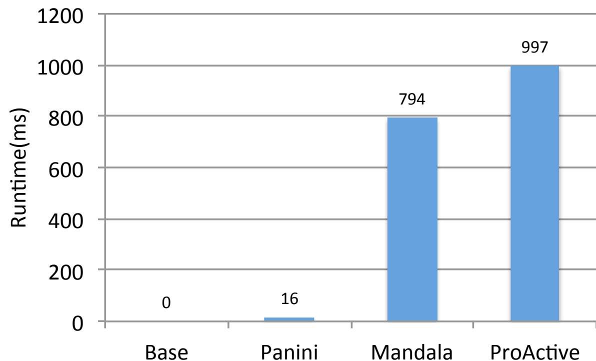

Figure 7. Claiming overhead of 10,000,000 future claims. The runtime of the base case is lesser than 1 millisecond.

The results shown in Figure 7 represents the mean of 10 steady state runtime in milliseconds. It is clear by comparing against the base case that using a future instead of a normal object indeed adds some amount of overhead when attempting to claim it. The result also shows that duck future has a much lower overhead than the other two. The performance difference is because Mandala and ProActive use reflection to obtain the result class to compare and check against the expected class, and again use reflection to invoke the actual method. In comparison, duck futures checks the completion of the future by a simple boolean check and directly calling the method on the actual object, adding a much smaller overhead to the process.

This result also shows that the claiming overhead is relatively insignificant compared to the overhead of generating the future, as the overall run time is much slower than the previous results from §4.3.1. Note that in this experiment 10 times more iterations were executed.

The results from both the performance experiments show that duck futures has an overall performance gain over Mandala and ProActive, with at least 54.85% performance improvement at 1,000,000 iterations using our benchmark program in §4.3.1. For the second benchmark in §4.3.2, the results also show that duck futures have a significantly lower overhead when claiming futures. From these two microbenchmarking experiments, we conclude that the result suggests that duck future's generative approach has a lower performance overhead than Mandala and ProActive.

4.3.3 Measuring the Call-Claim Distance of Futures

In this section, we use Boa to answer RQ4. For these experiments, we assume that each method call is replaced by calls using futures. By the wait-by-necessity rule, a result is used when a method invocation is invoked on the returned result. To measure the number of statements that are executed between making a method call and the result being used, we first mark each of the method calls whose result is assigned to a variable. Next, we count how many statements are declared, before we see an method invocation on the marked variable. Futures that escape the scope of the method by `return` statements are counted. If the result is never claimed in the same method, the distance is returned as ∞.

#	Occurrence	*All*		*All - (∞ + Returned)*	
		%	*Accum. %*	*%*	*Accum. %*
1	3,888,945	30.56%	30.56%	41.27%	41.27%
2	1,776,887	13.96%	44.52%	18.85%	60.12%
3	1,368,430	10.75%	55.27%	14.52%	74.64%
4	768,932	6.04%	61.31%	8.16%	82.80%
5	483,693	3.80%	65.11%	5.13%	87.93%
6	301,956	2.37%	67.49%	3.20%	91.14%
7	190,205	1.49%	68.98%	2.02%	93.16%
8	148,826	1.17%	70.15%	1.58%	94.74%
9	110,299	0.87%	71.02%	1.17%	95.91%
10	78,508	0.62%	71.63%	0.83%	96.74%
>10	307,301	2.41%	74.05%	3.26%	100.00%
∞	1,660,495	13.05%	87.10%	-	-
Returned	1,642,378	12.90%	100.00%	-	-
Total	12726855	100.00%	-	-	-

Figure 8. Distance between calls and claiming the results of the returned objects.

The result is shown in Figure 8. The distance of 1 means that the resulted object was used by the statement immediately after the method call. Of all the 12,726,855 method calls measured, 12.90% of them were returned by the method before being used, 13.05% were not used within the same method. Almost 40% (38.69%) of the results were claimed at a distance more than 4 statements away. Given that the average number of statements in methods using the same counting method is about 7 statements, this is a fairly sizable distance.

The result of this experiment suggests that most of the returned results are not being used immediately after the call, and applying transparent futures to programs is likely to gain some benefit from concurrency without modifying the program. Another observation is that a sizable percentage of objects leave the method without being claimed (potentially up to 25.95%). These are the cases where a refactoring approach to add futures to code may require a full interprocedural analysis to perform refactoring. For these cases, the transparency of duck futures can be very useful.

4.3.4 Applicability of Transparent Futures

In this section, we use Boa to answer RQ5, i.e. how frequently are classes defined by programmer not compatible with transparent futures? That is, classes for which a transparent future class cannot be created, or classes that will lead to incorrect code when used as a transparent future. As discussed in §4.1, there are 3 properties of classes we are looking for: final classes that cannot be subtyped, classes with final methods where we cannot implement the wait-by-necessity blocking mechanism, and non static public fields to which the blocking mechanism cannot be applied.

For the first experiment in attempt to answer this question, we first look at all the classes that are defined by the

programmer. We count the number of classes that have the above properties to find out how many of these classes are compatible with transparent futures.

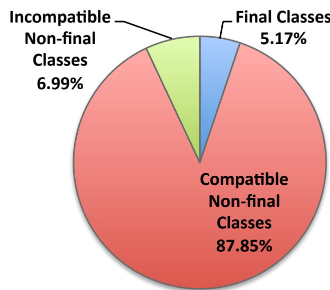

Figure 9. Percentage of final classes and transparent future compatible/incompatible non-final classes out of 8,244,248 public classes.

The percentage of final classes we found out is shown in Figure 9. Among 8,244,248 public classes, only 5.17% of the classes are declared as final. If we further look into the non final classes, we get the result shown in Figure 10.

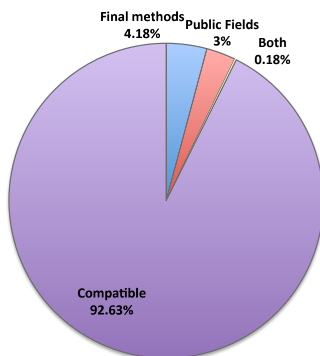

Figure 10. Percentage of classes with non static public fields or public final methods out of 7,818,244 non final classes.

Among all the public non final classes, 4.18% has final methods and non-public fields, 3.0% has public fields but not final methods, and 0.18% has both public fields and final methods. This leaves a 92.63% of non final classes that transparent futures are compatible with. From the results so far, it is clear that although the number of incompatible classes are not negligible, the majority of the classes are compatible with transparent futures.

To gather more data, we repeat the experiment on another set of classes in the open source projects. This time we only count the classes that are actually being used as return types within the same project they are defined. The results are shown in Figure 11 and Figure 12.

It is noticeable that classes used as return types more often have public fields and final methods in comparison to the overall set. The increase of public fields may be because these classes are more likely to be designed as data containers so programmers tend to want the contained data to be easily accessible. Another observation is the majority of classes are not being used as returned types, with 1,597,974 out of all 8,244,248 public classes (19.38%) being used as return types. Which also implies that only 19.38% of classes need duck futures generated.

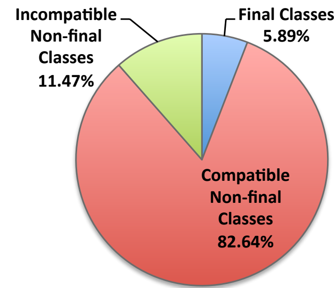

Figure 11. Percentage of final classes and transparent future compatible/incompatible non-final classes out of 1,597,974 public classes being used as return types.

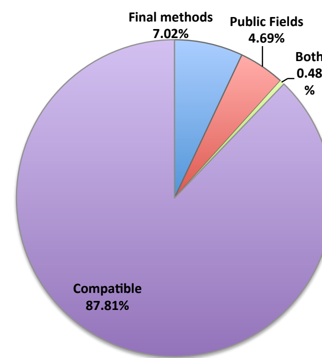

Figure 12. Percentage of classes with non static public fields or public final methods out of 94,082 non final classes.

For our last experiment to find out the applicability of transparent futures, we measure the percentage of different return types of all the non-private methods.

The result is shown in Figure 13. Among all the non-private methods, 50.36% are void methods, 22.28% of the methods return primitive types, and only 0.05% of the methods return final classes defined in the same project. We observed that in Java, the majority of methods return types other than classes. This result along with the previous result of incompatible classes are the minority of all classes imply that only a small portion of methods may return incompatible classes, and primitive types may be the biggest problem when using transparent futures. Note that the “other" category are not completely compatible classes. Incompatible

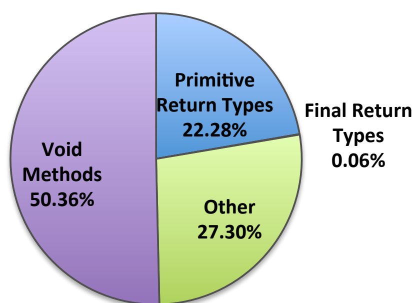

Figure 13. Percentage of different return types of non-private methods.

classes defined in libraries is included, and array types are also counted.

In conclusion, in our dataset, among all the classes defined in the project, 17.36% of the classes used as return types are incompatible with transparent futures. For the return types of methods, only at most 27.30% percent of non private methods return class types, with merely 0.06% of them being final classes from the same project. However, primitive types shows to be a major issue for transparent futures in Java.

4.4 Threats to Validity

We now outline the threats to our evaluation's validity.

4.4.1 Threats to Performance Evaluation

When measuring performance of Java and JVM-based techniques, it is hard to avoid other variables brought in by the tools implementing the techniques to affect the performance outcome. For example, the just-in-time (JIT) compilation overhead can affect the outcome greatly. To mitigate the effect of these variables, we used the runtime measurement methodology proposed by Georges *et al.*[14] to measure the performance in steady state.

4.4.2 Threats to Empirical Evaluation

Our call-claim analysis discussed in §4.3.3 is intraprocedural and is not weighted by the cost of statements. Therefore, a threat is that it pessimistically underestimates the cost of method calls. Furthermore, it also, pessimistically, counts only a single iteration of a loop. We believe this threat due to underestimation will not affect our conclusion, because adding complete call sequences will add more distance to the result.

In §4.3.4, our analysis excludes classes from imported libraries including the standard JDK. Although the corpus being examined also includes many libraries itself, this may slightly affect the precision of the result. In the future, we plan to use Boa to find out the most used libraries by projects, and perform an analysis using information of these libraries. Boa also plans to include the JDK library as one of their available datasets in the near future, which can allow us to further improve our analysis.

Finally, in all of our analyses we consider all projects as candidates to apply transparent futures. But in reality they may choose not to do so and may not gain any benefits from implicit concurrency.

5. Discussion

In this section we discuss the some of the aspects of evaluation transparent future models. For this paper, we use the following criteria borrowed from the work of Eugster [13].

Completeness: This criterion specifies the types that transparent futures can work with, and the proportions of the types that they can work on.

Safety: The safety criterion inspects additional actions not defined by the user introduced by the implicit transparent future process. The presence of additional actions may end up with side-effects or exceptions being thrown that is not expected by the user.

Overhead: This criterion measures the performance overhead introduced by transparent futures. In any approach, a future has to be created to be returned to the caller and an overhead is added. This includes the claim overhead of the future, where the future may have to check upon the completion of the execution of method. This aspect has already been specifically discussed in §4.

Transparency: This criterion represents the degree that the programmer are aware of a future being created for procedure calls. Transparency indicates the programmer is oblivious of the returned type and the asynchronous nature of the call.

5.1 Completeness

The completeness criterion specifies the types of Java, that can not be handled by the approaches and have a transparent future generated for the specified type. Approaches based on Java inheritance, generates futures by subtyping the expected return type. This strategy runs into trouble when subtyping is illegal. In Java, classes declared as final cannot be subtyped. For the same reason, arrays of any type and primitive types provided by Java proves difficult for inheritance based approaches to handle.

Another problem for inheritance based approaches is claiming of futures relies on overriding the methods of the subtyped class to provide the wait-by-necessity mechanism. Final methods and public classes that cannot be overridden or accessing does not go through methods bypasses the blocking mechanism and becomes a problem.

Duck futures implicitly make calls to incompatible classes synchronous as a workaround. Both Mandala and ProActive runs into the same problem. ProActive's approach is similar to ours. For these cases, method calls are simply implicitly

synchronous, and calling final methods and accessing public fields on a future leads to inconsistent behavior.

Mandala takes on a more restrictive but more consistent approach. Because methods in Java interfaces cannot be final, and all fields in interfaces are implicitly constant, using interfaces exclusively avoids having to deal with these difficulties when subtyping. Three limitations are applied by Mandala, the object being turned into a proxy must implement at least one interface, methods being invoked must be defined in the interface, and the return type of the method must be an interface.

5.2 Safety

It is important for a transparent future mechanism to not introduce potential execution of code that is not present in the original user defined program. This is to guarantee unexpected side effects will not happen. For many inheritance based approach, one vulnerability for this criteria comes from constructor of super classes. Constructors of Java classes are required to either explicitly invoke a constructor of its superclass as its first statement, or a no-argument constructor invocation will be added implicitly instead. This means that code that may be in the constructor of the superclass may compromise the safety criterion.

To avoid this problem, ProActive requires classes being used as return types to provide an empty constructor without any arguments. If such a constructor is not provided, the class is treated in the same way as a final class and invocations toward methods with such classes will be synchronous. However this does not guarantee that the constructor of the superclass higher up in the hierarchy does not invalidate safety.

The restriction of exclusively using interfaces by Mandala also removes the concern of constructors, because interfaces cannot define constructors and only the default constructor of the Object class is invoked.

5.3 Transparency

The Transparency criteria evaluates four different aspects: The return type of a call should appear to the caller as the expected type, and not a Future<T> type. This holds for duck futures and both works we compare with in this paper.

Identity revealing operations such as **instanceof** and == should be able to operate as if performed on the actual object. For example, a problem may occur when the program attempts to check the returned value against the **null** value.

The asynchronous nature of the future should be hidden. The future object can be passed around like normal objects, and the call should appear and behave the same as synchronous calls. An issue for asynchrony occurs for checked exceptions. A **try**/**catch** block on method invocations may not catch exceptions properly, as the control flow may have already exited the block when the exception is thrown. ProActive prevents this by making the calls that may throw exceptions synchronous. In addition, ProActive provides a semi-transparent solution so that the user can explicitly place blocking barriers around the **try**/**catch** block so that the control flow cannot exit the block until the execution of the method is completed.

In this section we have discussed the aspects of transparent future researchers are interested in. Approaches to implement transparent futures in Java using an inheritance strategy have many common challenges. Each work has their own decisions to overcome the challenge, but so far most of the approaches also have their own drawbacks and do not fully solves the problem. However, as we have shown in §4, some problems such as final classes may not be as major as it seems. These problems remains an open challenge, and await future researches to solve.

6. Related Work

Many approaches require manipulating futures manually [2, 17, 22]. Managing futures requires a good understanding of concurrency and its concerns in a multi-threaded execution context. To solve this problem transparent futures are introduced. The notion of futures-transparency is used to hide the complexity of futures to developers. A number of approaches provide fully-transparent futures [3, 8]. The goal of using fully transparent futures is to allow the use of legacy classes which were not designed in a concurrent context.

ProActive [8] uses inheritance to provide fully transparent futures. In ProActive, a transparent future is an instance of a class which extends the return value of the original method. For example, for a method call R m(...), a transparent future will be R' where **class** R' extends R. The subclass R' implements the wait-by-necessity behavior: until the real result is available the caller is blocked and once the result is available, every blocked caller is notified.

Mandala [3] approaches the problems by using interfaces. In Java, interfaces do not contain fields, methods of the interfaces are declared public, and interfaces cannot be declared public. Hence, all the inheritance related problems are solved by interface solution. But, interface constraints limits Mandala's applicability.

Researchers have found exception handling difficult with fully transparent futures [24]. Hence, an alternative solution to solve the problems is to not use fully transparent futures but use semi-transparent futures. Semi-transparent futures have strong typing and exceptions are always handled. In semi-transparent futures, developers are aware of the asynchronous nature of method invocations and have control over it. On the other hand, in Duck Futures managing of futures is completely oblivious to developers and developers do not have control over it.

Pratikakis *et al.* [19] proposed using proxies as futures to achieve transparency. The most significant difference between Duck Future and that of proxies is where the future is claimed. Their approach uses a static analysis to automat-

ically insert code to both generate and claim futures at the point of usage for the future.

Eugster [13] work also provides a proxy class for that extends the original class so it can be used as the original type. It provides an enhancement over previous dynamic proxy techniques on increasing the completeness of the proxies. The proxy class is generated at run-time as byte code when needed, and then loaded and linked. The **final** keyword and the problems it create are dealt with by treating final classes as non final classes at linking time, thus allowing class subtyping and overriding of methods. For public fields, field accesses are transformed to use setters and getters generated inside the proxy. Safety issues stemmed by constructors are handled by insertion of special constructors.

7. Conclusion

Creating and managing futures is tedious. Many researchers have worked on transparent futures to make creating and managing futures transparent to developers. Previous approaches create futures using reflection, which adds a substantial amount of overhead to runtime. In this paper we presented duck futures as transparent futures using a generative approach. The static approach reduces the overhead added to runtime, while providing the utility it needs to serve as transparent futures. Our work on duck futures shows that type information statically available is sufficient to create transparent futures. We show that as a result of being able to avoid reflection to dynamically create futures, duck futures has a lower runtime overhead compared to Mandala and ProActive, which heavily rely on reflection. We also performed a large scale study to show the potential benefit of applying transparent futures to synchronous code and gained a better understanding of the applicability of transparent futures.

For future work on duck futures, improvement on unsolved issues mentioned such as public constructors and exception is important going forward. It would also be interesting to see further empirical studies on applicability of duck futures, e.g. across different programming languages.

Acknowledgements

This work was supported in part by the NSF under grants CCF-14-23370, CCF-11-17937, and CCF-08-46059. We thank Steve Kautz for help and comments.

References

[1] Io: A small programming language. `http://www.iolanguage.com/`.

[2] JSR 166: Concurrency utilities. `http://www.jcp.org/en/jsr/detail?id=166`.

[3] VIGNERAS, P. Mandala. Web page, August 2004. `http://mandala.sf.net/`.

[4] Panini Website. `http://paninij.org/`.

[5] Dice Holdings, Inc. Sourceforge website 2015. `http://sourceforge.net/`.

[6] M. Bagherzadeh and H. Rajan. Panini: A concurrent programming model for solving pervasive & oblivious interference. In *Modularity'15*, 2015.

[7] D. Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[8] D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in Java. *Concurrency Practice and Experience*, 10(11-13):1043–1061, 1998.

[9] M. J. Compton. SCOOP: An investigation of concurrency in Eiffel. 2000.

[10] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *ICSE '13*, pages 422–431. IEEE Press, 2013.

[11] R. Dyer, H. Rajan, and T. N. Nguyen. Declarative visitors to ease fine-grained source code mining with full history on billions of AST nodes. In *GPCE '13*, pages 23–32, 2013.

[12] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen. Mining billions of AST nodes to study actual and potential usage of Java language features. In *ICSE 2014*, pages 779–790, 2014.

[13] P. Eugster. Uniform proxies for Java. In *OOPSLA '06*, pages 139–152. ACM, 2006.

[14] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA '07*, pages 57–76. ACM, 2007.

[15] R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[16] R. G. Lavender and D. C. Schmidt. Active object–an object behavioral pattern for concurrent programming. 1995.

[17] B. Liskov and L. Shrira. *Promises: linguistic support for efficient asynchronous procedure calls in distributed systems*, volume 23. ACM, 1988.

[18] D. A. Manolescu. Workflow enactment with continuation and future objects. In *OOPSLA '02*, pages 40–51, 2002.

[19] P. Pratikakis, J. Spacco, and M. Hicks. Transparent proxies for Java futures. In *OOPSLA '04*, pages 206–223. ACM, 2004.

[20] H. Rajan. Capsule-oriented programming. In *ICSE'15*, 2015.

[21] H. Rajan, S. M. Kautz, E. Lin, S. L. Mooney, Y. Long, and G. Upadhyaya. Capsule-oriented programming in the Panini language. Technical report, Iowa State University, August 2014.

[22] R. R. Raje, J. I. Williams, and M. Boyles. Asynchronous remote method invocation (ARMI) mechanism for Java. *Concurrency - Practice and Experience*, 9(11):1207–1211, 1997.

[23] E. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *OOPSLA '03*, pages 27–46, 2003.

[24] P. Vignéras. Transparency and asynchronous method invocation. In *On the Move To Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, pages 750–762. Springer, 2005.