Dynamically Composing Collection Operations through Collection Promises

Juan Pablo Sandoval Alcocer [1], Marcus Denker[2], Alexandre Bergel[1], Yasett Acurana[1]

[1]Department of Computer Science (DCC), University of Chile, Santiago, Chile
[2]INRIA- Lille Nord Europe, France

Abstract

Filtering, mapping, and iterating collections are frequent operations. It is known that composing a number of these operations may create intermediate collections causing an additional and unnecessary overhead. To reduce the number of intermediate collections it is often necessary to rewrite the source code and combine the operations. However, for some cases such reduction becomes aplicable only after a source code refactoring (*i.e.,* when the collection operations are in different methods) which could introduce code duplication.

In this paper we propose *Collection Promises* to dynamically compose collection operations in order to reduce the number of unnecessary intermediate collections. *Collection Promises* delay a number of collection operations and then merge them using compositions rules. By using *Collection Promises* developers can automatically reduce the intermediate collections even if the collection operations are in different methods.

1. Introduction

It is known that abstractions for data collections play a significant role in application performance. This situation is exacerbated in Pharo since loops and iterations are operations performed on collections. It has been shown that a considerable number of performance bugs and regressions are related with loops and collections [7, 8].

Composing collection operations involves the combination of collection operations, typically filtering, mapping, and iterating. For instance, consider the methods `on:` and `elementsNotEdge` in the *Roassal* application:

```
ROAdjustSizeOfNesting class>>on: element
  element elementsNotEdge do: [ :el |  ...].

ROElement>>elementsNotEdge
  ^ elements reject: #isEdge
```

Both methods realize a combined operation over a collection of elements. First, the method `elementsNotEdge` filters the collection, then the method `on:` iterates over the collection that results from the filter. Composing collection operations can introduce an additional overhead because each operation could create an intermediate and temporary collection. For instance, in the previous example the result of the `reject:` operation is temporary and intermediate. Creating these intermediate collections may lead to unnecessary overhead depending on the size of the collection, the number of composed collection operations, and the collection type.

Reducing the number of intermediate collections is a natural action to reduce the overhead. For instance, a composition of `reject:` and `do:` operations can be optimized by avoiding the intermediate collection, using the following method:

```
SequenceableCollection>>reject: rejectBlock thenDo: aBlock
  |  each |
  1 to: self size do: [ :index |
    (rejectBlock value: (each := self at: index))
      ifFalse: [ aBlock value: each ]].
```

The Pharo class `Collection` already provides a number of these "utility" methods to perform a combination of composed operations without using intermediate collections. Using these methods some optimizations can be applied replacing a sequence of calls to `reject:` and `do:` by a call to `reject:thenDo:`. However, in some cases this optimization becomes applicable only after a source code refactoring. For instance, to apply the method `reject:thenDo:` in our previous example, we need to join the methods `on:` and `elementsNotEdge` together, but we need to consider that the method `elementsNotEdge` is called in different methods. Therefore, these source code refactoring could produce code duplications.

This paper proposes *Collection Promises*, an alternative to composing collection operations that reduces the number of intermediate and temporary collections. Collection Promises delay a number of collection operations and then merge them using compositions rules. Composing collection operations through collection promises allow us to dynamically compose collection operations, even if the operations are in different methods.

We present an extension of the class `OrderedCollection` that supports promises. Our initial experiment that shows the overhead of using *Collection Promise* is almost insignificant compared to an "optimal" collection operation composition. Therefore, we conclude that *Collection Promise* is a good alternative to compose collection operations.

This paper is structured as follows: Section 2 illustrates the challenges to compose collection operations. Section 3 describes the concept of collection promises. Section 4 describes our initial prototype. Section 5 presents our initial experiment. Section 7 gives an overview of the related work. Section 8 concludes and presents future work.

IWST '16 August 23th, 2016, Prague, Czech Republic

ACM [to be supplied]. . . $15.00

2. Challenges

Inter-procedural Composed Collection Operations. Operations on collections may be performed across several methods. For instance, consider the following example:

```
RGBehaviorDefinition>>traits
  ^ self environment isRingObject
    ifTrue: [ self traitNames collect: [ :each | self environment
     traitNamed: each ] ]
    ifFalse:[ #() ]

RGBehaviorDefinition>>traitNames
  tokens := self traitCompositionSource parseLiterals flattened.
  ^tokens select: [:each | each first isUppercase].
```

The method `traitNames` executes two collection operations over the tokens collection (`select:` then `collect:`) and each one of these operations is performed in a particular method. The composed collection operation could therefore be optimized using the method `select:thenCollect:` that executes both operations without using an intermediate and temporary collection. However, we need to consider that the method `traitNames` could have different senders and the use of the method `select:thenCollect:` could introduce code duplication.

Detecting Temporary and Intermediate Collections. To apply this collection-related optimizations, it is necessary to detect temporary and intermediate collections. For instance, consider the following example of the *Morph* project.

```
MenuMorph>>detachSubMenu: evt
  | possibleTargets item subMenu index |
  possibleTargets := self items select:[:any| any hasSubMenu].
  possibleTargets size > 0 ifTrue:[
    index := UIManager default
        chooseFrom: (possibleTargets collect:[:t| t contents
     asString])
        title: 'Which menu?' translated.
    index = 0 ifTrue:[^self]].
  item := possibleTargets at: index.
  ...
```

Two collection operations are performed over the items collection. The collection resulting from the operation `select:` is stored in a temporary variable in order to use it in following instructions. Storing the collection resulting from an operation in a variable is a good indicator that the results may not be temporary and intermediate. However, there are cases where the collection result is stored in a variable and the result is still temporary and intermediate. For instance, consider the method `selectedCategories` of the class *CategoryWidget*.

```
CategoryWidget>>selectedCategories
  | associations |
  associations := self categoriesSelection associations select: [
      :assoc | assoc value ].
  associations := associations collect: [:assoc | assoc key ].
  ^ associations select: [:each | each notNil ]
```

The method `selectedCategories` has three composed collection operations. This method saves the result of the first `select:` operation in a variable, but this result is temporary and intermediate. The previous method could be improved as follows:

```
CategoryWidget>>selectedCategories
  | result associations |
  associations := self categoriesSelection associations.
  result := OrderedCollection new.
  associations do: [ :assoc|
        (assoc value and: [ assoc key notNil ]) ifTrue: [
          result add: assoc key.
        ].
  ].
  ^ result.
```

This new version of the method `selectedCategories` does not create intermediate collections. However, it could be more difficult to read and understand.

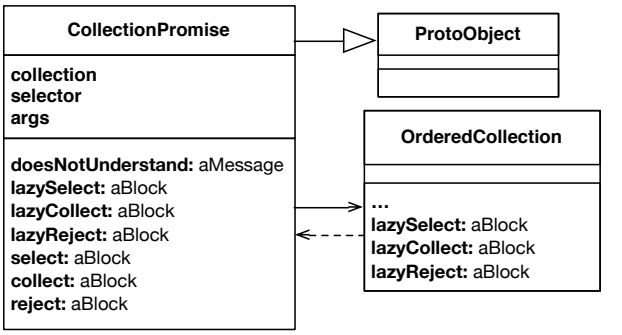

Figure 2. Implementation - Class Diagram

3. Collection Promise

We refer to a *Collection Promise* as an object which represents the collection result of an operation which is not yet computed. We propose the use of *Collection Promise* to delay the operations that create intermediate and temporary collections as long as possible, particularly, the operations `select:`, `collect:`, and `reject:`. A *Collection Promise* is computed when any other operation is executed.

The delayed operations are composed using composition rules to reduce the number of intermediate collections. For instance, consider the following contrived but representative example:

```
((students
  select: [ :each | each gpa > threshold ])
  collect: [ :each | each age ])
  median.
```

It contains a sequence of three collection operations `select:`, `collect:` and `median`. The first two operations could be delayed and combined to reduce the intermediate collection (*i.e.,* `select:thenCollect:`). The third operation forces the evaluation of the previous two operations because the result of the operation `median` is not a collection. Another common operation that could force the computation of a collection promise is `do:` which needs to iterate over the collection.

4. Implementation

This section presents a prototype of *Collection Promises*. The goal of our implementation is to support an initial experiment to see if the idea of *Collection Promises* is worthwhile, but the final implementation would need more work to compose a larger variety of combined operations.

Figure 2 shows the class diagram of our implementation. We add three methods to the class *OrderedCollection*: `lazySelect:`, `lazyCollect:`, `lazyReject:`. Instead of executing the operation and returning the collection result, these methods return an object of the class *CollectionPromise* which represents a promise to apply an operation in an ordered collection. In our implementation, we consider the operation `select:`, `collect:`, and `reject:` because these methods usually return a possible intermediate collection. For instance, the method `select:` is implemented as follows:

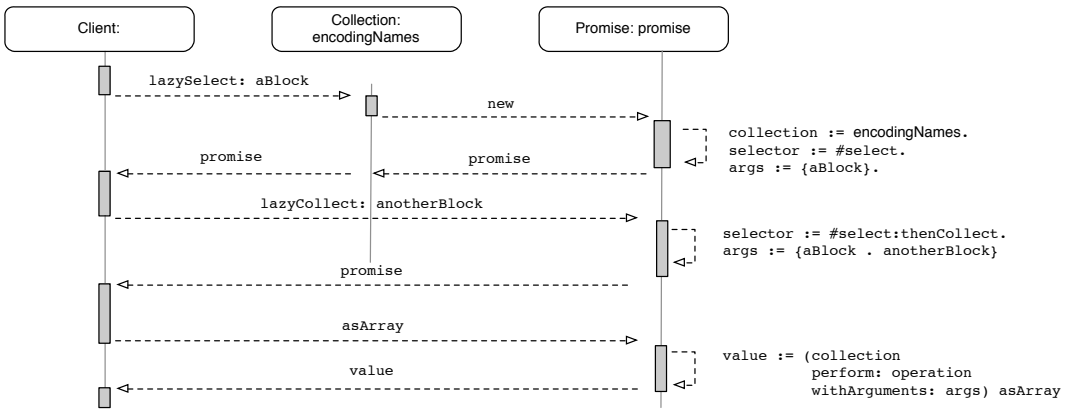

Figure 1. Composing Collection Operations using a Collection Promise

```
OrderedCollection>>lazySelect: aBlock
  ^ CollectionPromise new
      collection: self;
      selector: #select:;
      args: { aBlock };
      yourself.
```

To compose a collection operation promise the class `Collection Promise` has also the same methods `lazySelect:`, `lazyCollect:`, and `lazyReject:`. These methods have a number of rules to compose the promises without intermediate and temporary collections. For instance, consider the implementation of the method `lazySelect:`:

```
CollectionPromise>>lazySelect: aBlock
  "... composition rules ..."
  (self selector = #select:) ifTrue:[
    |arg|
    arg := self args first.
    self args: {[ :ele | (arg value: ele) and: [aBlock value:ele]]}.
    ^ self.].
  (self selector = #collect:) ifTrue:[
    self selector: #collect:thenSelect:.
    self args: {args first . aBlock}.
    ^ self].
  "... if none of the rules could be applied ..."
  self collection: (self evaluate).
  self selector: #select:.
  self args: { aBlock }.
  ^self.
```

This method contains two rules, the first one composes two `select:` operations and the second one composes a `collect: thenSelect:` operation. If none of these rules can be applied, then the promise that receives the message is evaluated and the method returns a `select:` promise over the resulted collection.

If a promise receives a message that can not be delayed, then the promise is evaluated and the message is sent to the collection that results from the promised operations.

```
CollectionPromise>>doesNotUnderstand: aMessage
  ^ self evaluate perform: aMessage selector withArguments:
      aMessage arguments.
```

A collection promise also supports `select:`, `collect:` and `reject:` operations. Unlike the lazy operations, the results of these operations are expected to be collections and not promises. An easy solution to support these methods is to evaluate the promise and then apply the operation over the resulting collection, but we could miss an opportunity to compose the operation. Therefore, if a promise receives a `select:`, `collect:`, or `reject:` operation, we first try to compose these operations by calling their `lazy` equivalent method, then we evaluate the promise and return the result. For instance, consider the implementation of the method `select:` on the class `CollectionPromise`:

```
CollectionPromise>>select: aBlock
  ^ (self lazySelect: aBlock) evaluate.
```

This method first calls to the method `lazySelect:` in order to apply the rules to compose the operations by using promises, then evaluates the composed promise.

To use our implementation of collection promises we need to call the methods `lazySelect:`, `lazyCollect:`, and `lazyReject:`. For instance, consider the following code of the *Greace* project that we adapt to use collection promises.

```
GRCode class>allCodecs
  ^ self subclasses
      inject: self codecs asArray
      into: [ :result :each | result , each allCodecs ]

GRPharoGenericCodec>>codecs
  ^ (TextConverter allEncodingNames
      lazySelect:  [ :each |  self supportsEncoding: each ])
      lazyCollect: [ :each | self basicForEncoding: each greaseString
       ]
```

We replace the collection operations `select:` and `collect:` in the method `codecs` by `lazySelect:` and `lazyCollect:` respectively. Figure 1 illustrates the execution of the expression `self codecs asArray`. By using collection promises in this particular example, the results of the operation `lazySelect:` is a collection promise, and the result of the operation `lazyCollect:` is a promise of a composed operation `select:thenCollect:`. The promised operations are executed when a method `asArray` is executed, because in our implementation the method `asArray` does not support promises. Therefore the method `doesNotUnderstand:` is executed, which computes the promise.

Note that the result of the operation `select:` is no longer needed because it is temporary and intermediate, then it is not necessary to create a new promise object of the `collect:` operation. We can reuse the object of the `select:` promise and change its state to a `select:thenCollect` promise. By reusing the object of the `select:` promise we assume that the collection promise is temporary and intermediate. If a collection promise is not temporary or intermediate, it could introduce a side effect, causing an unexpected

behavior at run-time. Therefore, we need to analyze which collection operations (*i.e.,* `select:`, `collect:` and `reject`) could be safely replaced by lazy operations (*i.e.,* `lazySelect:`, `lazyCollect:`, and `lazyReject:`) through collection promises. The analysis and the replacement could be done by using a dedicated profiler or during the compilation process.

Table 1 lists the composed collection operations that our implementation supports. Our current implementation composes the operations on demand and in pairs. For instance, to compose a sequence of three operations, it attempts to compose the first two operations and then composes the result with the third operation. Therefore, the composition could change depending on the order of the operations. As future work, we plan to support a larger variety of collection operations and rules to combine more than two operations.

5. Case Study

Baseline for comparison. We compare three different ways to perform a sequence of collection operations:

- **With Intermediate Collections**, using a combination of the methods `select`, `collect`, and `reject`.
- **With Collection Promises**, using a combination of the methods `lazySelect:`, `lazyCollect:`, and `lazyReject:`.
- **Without Intermediate Collections**, manually avoiding the the intermediate collection, for instance, using the method `select:thenCollect:` directly.

Micro-benchmarks. We define a set of micro-benchmarks and compare six pairs of collection operations. For instance, the benchmarks for the operations *select then select* are:

```
benchWithIntermediateCollections
   ((collection select:#even)  collect:[:x | x + 1]) size
benchManuallyOptimized
   (collection select:#even thenCollect:[:x| x + 1]) size
benchWithPromises
    ((collection lazySelect:#even) lazyCollect:[:x | x + 1]) size.
```

Table 1 summarizes all the combinations that we compare.

Results. We run the benchmarks 25 times with a previous warm-up session. For this experiment we use a collection of 10,000,000 numbers. Table 1 gives the execution time of each benchmark. It also shows the error margin with a confidence interval of 90%. As expected, the results show that the benchmarks that use intermediate collections run slower than the benchmarks that do not use intermediate collections. We also see that the benchmarks that use promises are slower than the manually optimized benchmarks. However, the benchmarks that use promises are considerable faster than the benchmarks that use intermediate collections.

Collection Size. The results of the table 1 can change depending on the collection size. To better understand how the collection size affects our experiments we measure the execution time of the benchmark *select:thenCollect:* using three collections of size 1,000; 100,000 and 10,000,000 respectively. The results in Figure 3 show that the difference between executing composed collection operations is most notorious with bigger collections. Figure 3 also shows that composed collection operations using promises introduce almost the same overhead than the manual (and "optimal") composition.

Table 1. Micro-benchmarks Comparison. (w/=with , w/o = without)

	w/ Intermediate Collection	w/ Collection Promises	w/o Intermediate Collection
select then select	2115ms +/-21ms	1696ms +/-15ms	849ms +/-9.1ms
collect then collect	1164ms +/-19ms	724ms +/-6.9ms	628ms +/-34ms
collect then select	2015ms +/-37ms	1520ms +/-13ms	1492ms +/-16ms
reject then collect	1712ms +/-17ms	1484ms +/-16ms	1483ms +/-14ms
select then collect	1669ms +/-24ms	1503ms +/-14ms	1465ms +/-15ms
collect then reject	1963ms +/-29ms	1893ms +/-11ms	1886ms +/-16ms

6. Discussion

By using collection promises we can reduce the overhead associated to create and fill unnecessary intermediate collections, but we need to consider that the use of collection promises also introduces an overhead.

In the previous section, we have shown that (for a particular case study) the overhead associated to create and fill intermediate collections is greater than the overhead associated to use collection promises, particularly with big collections. However, there are a number of scenarios where the use of collection promises could lead to a small performance regression. For instance, if we delay a collection operation that will not be composed in the future or if the operations are done over empty collections. As future work, we plan to automatically identify these scenarios in order to avoid small performance regressions.

7. Related Work

There are several works that improve the queries in relational databases and objects [1]. A number of approaches have been proposed for a closer integration of queries in the programing language [6, 9]. For instance, *LINQ* [4, 5], *LINQ* reifies queries on collections to analyze and optimize them. These queries can be executed in a variety of backends, for instance, SQL databases. There are also many approaches to integrate SQL queries into programs, such as HaskellDB [5] or Ferry [3].

Similar to our work, Giarrusso *et al.* introduce *SQUOPT* a deep embedding of a version of the Scala collections API which reifies queries and can optimize them at run-time [2]. *SQUOPT* reifies the collection queries syntactic structure as expression trees to finally optimizing them. To use *SQUOPT* it is necessary to slightly adapt the code in a similar way to our work.

Our work focuses on improving the way that collection operations are composed along software execution in dynamically typed object-oriented programs, which is the case of Pharo. Compared to these works, we have already implemented few collection-related optimizations, as future work we plan to improve our work by applying more advanced optimizations.

8. Conclusion

We have introduced the challenges to efficiently compose collection operations in dynamically typed object-oriented languages. We propose *Collection Promise* as an alternative solution to dynamically compose collection operations. Thanks to *Collection Promises* it is possible to compose collection operations even if they are in different methods. We present a small experiment that shows that the overhead of using *Collection Promise* is almost insignificant compared to an "optimal" collection operation composition. As future work we plan to: 1) automatically detect temporary and intermediate collections, and 2) support a larger variety of collections and operations.

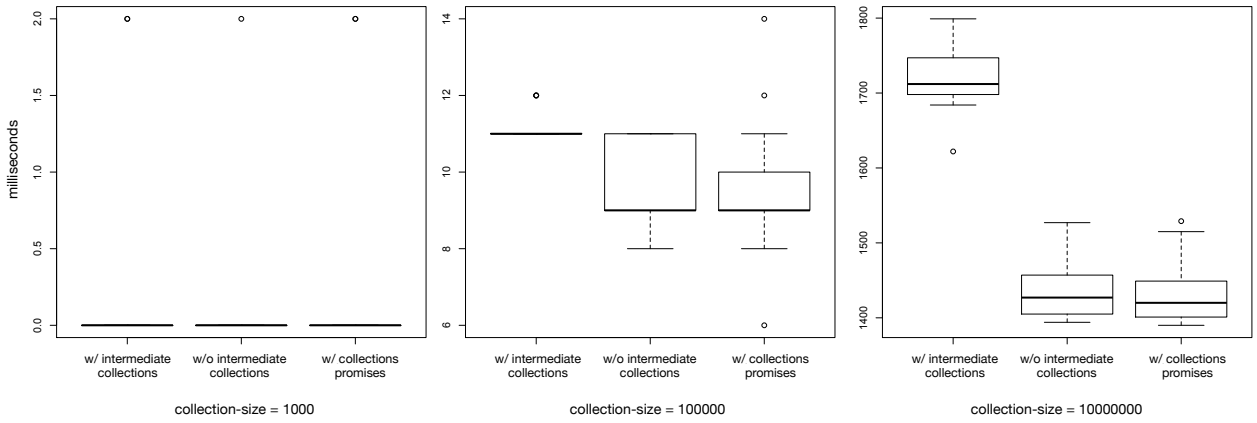

Figure 3. Composing Collection Overhead vs Collection Size (w/=with , w/o = without)

Acknowledgments

Juan Pablo Sandoval Alcocer is supported by a Ph.D. scholarship from CONICYT, Chile, CONICYT-PCHA/Doctorado Nacional para extranjeros/2013-63130199.

References

[1] Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25(4):457–516, December 2000.

[2] Paolo G. Giarrusso, Klaus Ostermann, Michael Eichberg, Ralf Mitschke, Tillmann Rendel, and Christian Kästner. Reify your collection queries for modularity and speed! In *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development*, AOSD '13, pages 1–12, New York, NY, USA, 2013. ACM.

[3] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: Database-supported program execution. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 1063–1066, New York, NY, USA, 2009. ACM.

[4] Language Integrated Queries. http://plone.org/products/archgenxml.

[5] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706, New York, NY, USA, 2006. ACM.

[6] V. K. S. Nerella, S. Madria, and T. Weigert. Efficient caching and incrementalization of object queries on collections in programming codes. In *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*, pages 229–238, July 2014.

[7] Juan Pablo Sandoval Alcocer and Alexandre Bergel. Tracking down performance variation against source code evolution. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, pages 129–139, New York, NY, USA, 2015. ACM.

[8] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. Learning from source code history to identify performance failures. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 37–48, New York, NY, USA, 2016. ACM.

[9] Darren Willis, David J. Pearce, and James Noble. Efficient object querying for java. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 28–49, Berlin, Heidelberg, 2006. Springer-Verlag.