Deterministic Lazy Mutable OCL Collections

Edward D. Willink

Willink Transformations Ltd, Reading, England,
ed_at_willink.me.uk

Abstract. The Collection iterations and operations are perhaps the most important part of OCL. It is therefore important for an OCL evaluation tool to provide efficient support for Collections. Unfortunately, some clauses of the OCL specification appear to inhibit efficient or deterministic support. We review the inhibitions and demonstrate a new deterministic and lazy implementation that avoids them.

Keywords: OCL, collection, deterministic, lazy, mutable

1 Introduction

The OCL specification [11] defines an executable specification language suitable for use with models. OCL's power comes from its ability to evaluate characteristics of multiple model elements using iterations and operations over collections.

The side-effect free functional characteristics of OCL should provide excellent opportunities for optimized evaluation, but sadly the optimization in typical OCL tools is poor. Collection evaluation is an area that should be particularly good, however it is very easy for the efficiency and/or memory usage to be outstandingly bad.

Deterministic execution is a desirable property of any language; very desirable if you are attempting to debug an obscure failure. Unfortunately today's OCL tools are not deterministic and so OCL-based Model-to-Model transformations tools also lack determinism.

In Section 2, we review the problems that the OCL specification appears to pose. In Section 3 we revisit these problems to identify over-enthusiastic or inappropriate reading of the OCL specification. Then in Section 4 we introduce our new Collection implementation that solves the problems. The new solution is still work in progress and so in Section 5 we describe what remains to do to integrate it effectively. In Section 6 we look at related work and conclude in Section 7.

2 The Problems

We briefly review some implementation challenges that the OCL specification provides.

Collection types: Four concrete derivations of the abstract *Collection* type are specified to support the four permutations of ordered/not-ordered, unique/not-unique content. These derived collections types are

- *Bag* - not-ordered, not-unique
- *OrderedSet* - ordered, unique
- *Sequence* - ordered, not-unique
- *Set* - not-ordered, unique

Java implementations may use a custom class, `LinkedHashSet`, `ArrayList` and `HashSet` respectively to implement these four `Collection` kinds.

Problem: Four distinct collection types.

Immutability: OCL is a functional language free from side effects. It is therefore impossible to modify an OCL *Collection*. There are no operations such as `Set::add(element)` that modify the receiver. Rather there are operations such as `Set::including(element)` that return a new `Set` based on the receiver and including the additional `element`. The obvious implementation of a cascade of operations such as `a->including(b)->including(c)->including(d)` creates a new intermediate collection between each operation.

Problem: Immutability implies inefficient collection churning.

Eagerness: OCL operations are defined as a computation of an output from some inputs. A cascade of operations such as `a->including(b)->excludes(c)` is therefore evaluated in three steps as get-`a`, then create `a+b`, and finally test `a+b` for `c` content. There is no mechanism for early discovery of a `c` to bypass redundant computations.

Problem: Specification implies eager evaluation.

Invalidity: A malfunctioning OCL evaluation does not throw an exception, rather it returns the `invalid` value, which will normally be propagated through invoking computations back to the caller. However OCL has a strict Boolean algebra that allows the `invalid` value to be 'caught' when, for instance, ANDed with the `false` value. The presence of the `invalid` value in a collection is prohibited, or rather the whole collection that 'contains' the `invalid` value is replaced by the `invalid` value. The result of a collection evaluation cannot therefore be determined until every element is present and checked for validity.

Problem: Invalidity implies full evaluation.

Determinism: Each collection type has distinct useful capabilities and so conversions between collection types are specified to facilitate their use. However, when the `asOrderedSet()` and `asSequence()` operations are applied to not-ordered collections, the operations must create an ordering without any clue as to what a sensible ordering criterion might be. This is obviously impossible and so typical Java implementations use the indeterminate order provided by a Java iteration over an underlying Java `Set`.

Problem: `asOrderedSet()` and `asSequence()` imply indeterminacy.

OCL equality: OCL is a specification language and when dealing with numbers, OCL uses unbounded numbers. Consequently the following OCL expressions are true:

```
1 = 1.0  Set{1,1.0}->size() = 1  Set{Set{1},Set{1.0}}->size() = 1
```

When using Java to implement OCL, the numeric equality is satisfied by the primitive types `int` and `double` but not by the object types `Integer` and `Double`. Since Java sets use object equality to establish uniqueness, a naive implementation may malfunction if it assumes that OCL and Java equality are the same.

Problem: OCL and Java equality semantics are different.

3 The Problems Revisited

The foregoing problems lead to poor and even inaccurate OCL implementations. We will therefore examine them in more detail to distinguish myth and truth before we introduce our new solution.

3.1 Immutability

While OCL may provide no operations to modify Collections, it does not prohibit modification by underlying tooling. A modification that does not affect OCL execution is permissible.

An evaluation of `a->including(b)->including(c)` may therefore re-use the intermediate collection created by `a->including(b)` and modify it to create the final result. This is safe since the intermediate result cannot be accessed in any other way than by the subsequent `->including(c)`. If there are no other accesses to `a`, it is permissible to modify `a` twice and avoid all intermediates.

3.2 Eagerness

While the specification may imply that evaluations should be performed eagerly, this is just the way specifications are written to ease understanding. An implementation is permitted to do something different so long as the difference is not observable. Lazy evaluation is a tactic that has been used with many languages. OCL has a strong functional discipline and so laziness has much to offer in an OCL evaluator. Unfortunately OCL development teams have been slow to exploit this tactic.

3.3 Invalidity

The OCL specification is far from perfect. In OCL 2.0, there were the three overlapping concepts of *null*, *undefined* and *invalid*. OCL 2.2 clarified the concepts by eliminating *undefined* and so distinguished *null* and *invalid*, but *invalid* is still inadequate to represent real execution phenomenon.

There is currently no distinction between program failures such as

– divide by zero
– *Sequence*/*OrderedSet* index out of range
– *null* navigation

and machine failures such as

– stack overflow
– network failure

Since machine failures are not mentioned by the specification, it would seem that they must be `invalid`, but only very specialized applications such as the OCL specification of a debugger can be expected to handle machine failures. Consequently the treatment of machine failures as `invalid` for the purposes of 4-valued (`true`,`false`,`null`,`invalid`) strict logic evaluation seems misguided. Rather a further fifth `failure` value for machine failure should be non-strict so that machine failures are not catchable by logic guards. The fourth strict `invalid` value should apply only to program failures.

Program failures are amenable to program analysis that can prove that no program failure will occur. When analysis is insufficiently powerful, the programmer can add a redundant guard to handle e.g. an 'impossible' divide-by-zero. With 5-valued logic we can prove that the partial result of a collection evaluation will remain valid if fully evaluated and so avoid the redundant full calculation when the partial calculation is sufficient.

Proving that null navigations do not occur is harder but an analysis of null safety is necessary anyway to avoid run-time surprises [5].

Once machine failures are irrelevant and the absence of program failures has been proved, a partial collection result may be sufficient; the redundant evaluations can be omitted.

3.4 Determinism

Determinism is a very desirable characteristic of any program evaluation, particularly a specification program. Is OCL really non-deterministic?

`Collection::asSequence()` is defined as returning elements in a collection kind-specific order.

The `Set::asSequence()` override refines the order to *unknown*, which is not the same as *indeterminate*.

The `Collection::any()` iteration specifies an *indeterminate* choice between alternatives.

The foregoing appears in the normative part of the specification. Only the non-normative annex mentions a lack of determinism for order discovery.

It is therefore unclear from the specification text whether an OCL implementation of order discovery may be non-deterministic. A clarified OCL specification could reasonably take either alternative. If order discovery is deterministic, it is easy for `Collection::any()`'s choice to be consistent with that discovery.

In practice, typical OCL implementations use a Java `Set` to realize OCL *Set* functionality. The iteration order over a Java `Set` depends on hash codes,

which depend on memory addresses, which depend on the unpredictable timing of garbage collection activities. It is therefore not possible for typical OCL implementations to be deterministic.

It would appear that implementation pragmatics are driving the specification or at least the user perception of the specification. But indeterminacy is so bad that it would be good to find a way to make OCL deterministic.

3.5 Four Collection types

The four permutations of unique and ordered provide four collection behaviors and four specification types, but do we really need four implementation types? With four types we may have the wrong one and so we need conversions. UML [10] has no collection types at all. What if an implementation realized all four behaviors with just one implementation type? One benefit is obvious; no redundant conversions.

4 New Collection Solution

Our new solution has only one *Collection* implementation type that exhibits all four *Collection* behaviors, but only one at a time. To avoid confusion between our new *Collection* implementation and the OCL abstract *Collection* or the Java `Collection` classes, we will use `NewCollection` in this paper[1].

4.1 Deterministic Collection Representation

A `NewCollection<T>` instance uses two Java collection instances internally.

- `ArrayList<T>` of ordered elements.
- `HashMap<T,Integer>` of unique elements and their repeat counts.

For a *Sequence*, the `ArrayList` serializes the required elements; the `HashMap` is unused and may be `null`.

For a *Set*, the keys of the `HashMap` provide the unique elements each mapped to a unit `Integer` repeat count; the `ArrayList` serializes the unique elements in a deterministic order.

For an *OrderedSet*, the keys of the `HashMap` provide the unique elements each mapped to a unit `Integer` repeat count; the `ArrayList` serializes the unique elements in the required order.

For a *Bag*, the keys of the `HashMap` provide the unique elements each mapped to a repeat count of that element; the `ArrayList` serializes the unique elements in a deterministic order.

The Java implementation of a `HashSet` uses a `HashMap` and so using a `HashMap` for *Set* and *OrderedSet* incurs no additional costs. On a 64 bit machine, each `HashMap` element incurs a 44 byte cost per `Node` and typically two 8

[1] The Eclipse OCL class name is currently LazyCollectionValueImpl

byte costs for pointers. Using an `ArrayList` as well as a `HashMap` increases the cost per entry from 60 to 68 bytes; a 13% overhead for non-*Sequence*s.

Use of an `ArrayList` to sequence the unique elements allows an efficient deterministic iterator to be provided for all kinds of *Collection*.

Since a *Set* now has a deterministic order, there is no implementation difference between a *Set* and an *OrderedSet*.

The deterministic order maintained by the `ArrayList` is based on insertion order. New elements are therefore added at the end or not at all, which avoids significant costs for `ArrayList` maintenance.

For a *Bag*, there is a choice as to whether an element iteration is over all elements, repeating repeated elements, or just the unique elements. The `NewCollection` therefore provides a regular `iterator()` over each element, and an alternative API that skips repeats but allows the repeat count to be accessed by the iterator. *Bag*-aware implementations of *Collection* operations can therefore offer a useful speed-up.

The `NewCollection` supports all *Collection* behaviors, but only one at a time. Non-destructive conversion between behaviors can be performed as no-operations. A *Set* converts to a *Sequence* by continuing to use the `ArrayList` and ignoring the `HashMap`. However the conversion from a *Sequence* to a *Bag* or *Set* requires the `HashMap` to be created and non-unique content of the `ArrayList` to be pruned; a new `NewCollection` is therefore created to avoid modifying the original `NewCollection`.

The `NewCollection` does not inherit inappropriate Java behavior. The problems with inconsistent OCL/Java equality semantics can therefore be resolved as `NewCollection` delegates to the internal `HashMap`.

4.2 Performance Graphs

The performances reported in the following figures use log-log axes to demonstrate the relative linear/quadratic execution time behaviors over a 6 decade range of collection sizes. The measurements come from manually coded test harnesses that instrument calls to the specific support routines of interest. Considerable care is taken to ensure that the 64 bit default Oracle Java 8 VM has warmed up and is garbage free. Curves are 'plotted' backwards i.e. largest collection size first to further reduce warm up distortions. Each plotted point comes from a single measurement without any averaging. Consequently the occasional 'rogue' point is probably due to an unwanted concurrent activity and demonstrates the probable accuracy of surrounding points even at the sub-millisecond level. Genuine deviations from smooth monotonic behavior may arise from fortuitous uses of L1 and L2 caches. Garbage collection may lead to inconsistent results for huge collection sizes.

4.3 Deterministic Collection Cost

Fig 1 shows the time to create a *Set* from a *Sequence* of distinct integers, contrasting the 'old' Eclipse OCL *Set* with the 'new' NewCollection *Set*. Overall

the ‘new’ design is about 2 times slower corresponding to the use of two rather than one underlying Java collection.

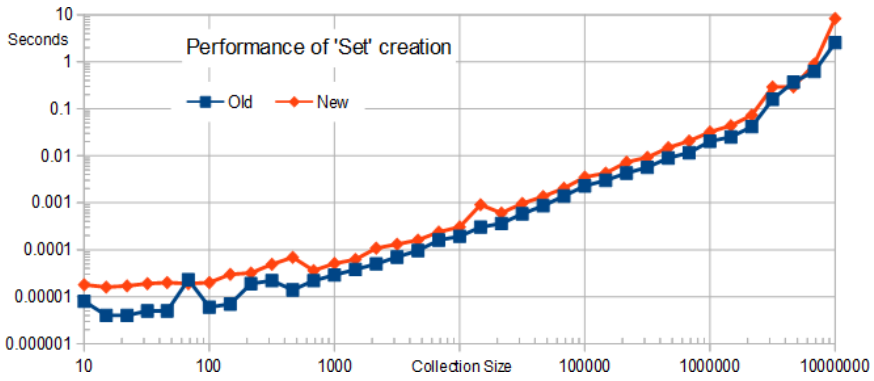

Fig. 1. ‘Set’ Creation Performance

A corresponding contrast of iteration speed is shown in Fig 2. The ‘new’ design is now about three times faster since the iteration just traverses adjacent entries in the deterministic `ArrayList` rather than the sparse tree hierarchy of non-deterministic `HashMap` nodes.

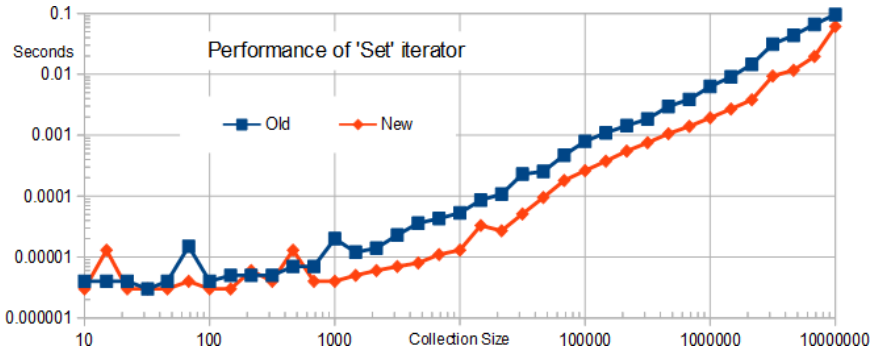

Fig. 2. ‘Set’ Iteration Performance

Iteration is faster than creation and so it depends how often the *Set* is used as to whether ‘new’ or ‘old’ is faster overall. More than three uses and the ‘new’ design is faster as well as deterministic. Even when used only once the speed penalty is less than a factor of two. Determinism is therefore practical and incurs acceptable size and speed costs.

4.4 Lazy Usage

The 'eager' exposition of `NewCollection`'s `ArrayList` solves the problem of indeterminacy. The lazy use of a `HashMap` as well as the `ArrayList` supports conversions and non-*Sequence* collections.

The `NewCollection` may also be be used for lazy evaluation by providing careful support for Java's `Iterator` and `Iterable` interfaces.

When a `NewCollection` has a single consumer, its `Iterator` may be used directly by invoking `iterator()` to acquire an output iterator that delegates directly to the input.

When a `NewCollection` has multiple consumers, it must be used as an `Iterable` to provide a distinct `Iterator` for each consumer. `iterable()` is invoked to activate the caching that then uses an internal iterator to iterate over the input at most once.

Considering: `a->including(b)->including(c)`

An eager implementation of `Collection::including` might be implemented by the `IncludingOperation.evaluate` method as shown in Fig 3.

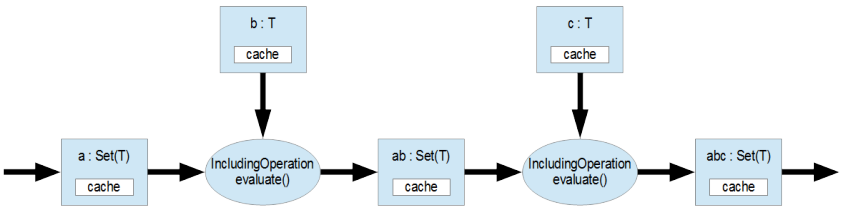

Fig. 3. Example Eager Evaluation Data Flow

The stateless `IncludingOperation::evaluate()` eagerly accesses the `a` and `b` values cached by their Variable objects and creates the intermediate `ab`. A second `IncludingOperation::evaluate()` similarly produces the result `abc`. Three collection caches are fully populated for each of `a`, `ab` and `abc`.

The lazy implementation shown in Fig 4 uses an `IncludingIterator` object that has a `current` iteration context. The iterator iterates to produce the required output, one element at a time by fetching the inputs one element at a time and interleaving the additional value at the correct position. No computation is performed until an attempt is made to access the `abc` result. Since the result cache is missing, the `abc` access invokes `IncludingIteration::next()` to provide each element of `abc` that is required. `IncludingIterator::next()` provides its result from `c` or by invoking `next()` on `ab`, which in turn acquires its values from `a` or `b`. No input, intermediate or output collection caches are

required; `a` can read its source one element at a time, `ab` relays its values one at a time, and the `abc` output may be accessed one element at time. This is a major size improvement, three uncached `NewCollections` that relay one element at a time, rather than three fully-cached `NewCollections`.

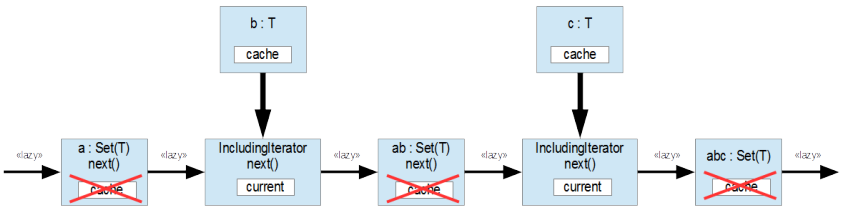

Fig. 4. Example Lazy Evaluation Data Flow

If `a` or `c` has multiple consumers, as shown in Fig 5, the undesirable repetition of the lazy including computations is avoided by activating caches where the multi-use occurs. This is slightly awkward to implement since the first consumer must invoke `NewCollection.iterable()` to activate the cache before any consumer invokes `NewCollection.iterator()` to make use of the collection content. As part of a general purpose library used by manual programmers this programming discipline could cause many inefficiencies. However as part of an OCL tool, an OCL expression is easily analyzed to determine whether a collection variable is subject to multiple access. If analysis fails, `iterable()` can be invoked just in case.

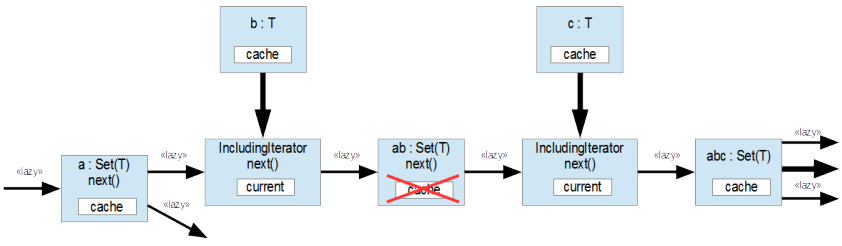

Fig. 5. Example Lazy Cached Evaluation Data Flow

Eager, lazy and cached evaluations share the same structure of operation and variable interconnections. The correct behavior is determined by analysis of the OCL expression. For a singly accessed collection, a transparent behavior is configured. For multiple access, a cached behavior is configured in which the source iteration is lazily cached for multiple use by the multiple accesses. Unfortunately, collection operations, such as `Collection::size()`, are unable to return a result until the source has been fully traversed and so an eager evaluation is sometimes unavoidable.

4.5 Lazy Cost

In Fig 6 we contrast the performance of eager and lazy OCL evaluation of the inclusion of two values into a *Sequence* of *Integer*s.

Fig. 6. Double Including 'Sequence' Performance

For more than 1000 elements, the top curve shows the lazy approach scaling proportionately. The next curve shows the aggregate performance of eager evaluation also scaling proportionately until garbage collection affects results at 10,000,000 elements. The bottom two curves show the contributions to the aggregate from the eager evaluation, and the final result traversal.

For small *Sequence*s with fewer than 1000 elements, the higher constant costs of the eager approach dominate and the lazy approach is perhaps five times faster.

For larger *Sequence*s, the lazy approach is about two times slower since an outer element loop traverses iterations for each partial computation whereas the eager approach has tighter inner loops for each partial computation.

For the largest 10,000,000 element result, garbage collection has started to affect the eager evaluation with its three full size collection values for input, intermediate and output. In contrast, the lazy approach only uses a few hundred bytes regardless of model size and so is much less affected by huge models.

The lazy approach is clearly superior with respect to memory consumption, and also faster for up to about 1000 elements. For larger sequences, lazy evaluation may be two times slower. Since lazy evaluation offers the opportunity to skip redundant computations, we may conclude that in the absence of application-specific profiling measurements, lazy evaluation should be used.

4.6 Mutable Collections

As suggested above, lazy evaluation is not always better. The simple example in Fig 4 replaced three fully-cached by three uncached `NewCollection`s but also introduced two intervening `IncludingIterator` objects. Invocation of `next()` to return an output object traverses the lazy sources incurring four nested invocations of `next()`. For an iteration such as

```
aCollection->iterate(e; acc : Set(String) |
    acc->including(e.name))
```

the overall iterate of an N-element `aCollection` evaluates using a chain of N interleaved `NewCollection` and `IncludingIterator` objects. The overall evaluation incurs a quadratic $2 * N * N$ cost in `next()` calls.

Of course the traditional approach of creating a new Collection for each invocation of `including` also incurs a quadratic cost through creating and copying N collections of approximately N-element size.

In order to achieve a more reasonable cost we can use a non-OCL mutable operation behind the scenes:

```
aCollection->iterate(e; acc : Set(String) |
    acc->mutableIncluding(e.name))
```

This exploits the invisibility of the intermediate values of `acc`. The evaluation should therefore analyze the OCL expression to detect that the single use of `acc` allows the immutable `including()` operation to be evaluated safely and more efficiently using the internal `mutableIncluding()` operation.

In Fig 7 we contrast the performance of the accumulation that computes

`S->iterate(i; acc : C(Integer) = C{} | acc->including(i))`

using `Set` or `Sequence` as the `C` collection type and `C{1..N}` as the `S` source value for an `N` collection size.

The new approach uses mutable evaluation to re-use `acc` and so avoid churning. The old approach uses the one new *Set* churn per `Set::including` execution as currently practiced by Eclipse OCL [7] and USE [12] (Dresden OCL [6] creates two *Set*s). The new approach scales linearly and so is clearly superior to the traditional quadratic cost. The new approach has a two-fold cost for using *Set*s rather than *Sequence*s; much less than when churning occurs.

Note that this optimization relies on a 'compile-time' OCL expression analysis that replaces `including` by `mutableIncluding`.

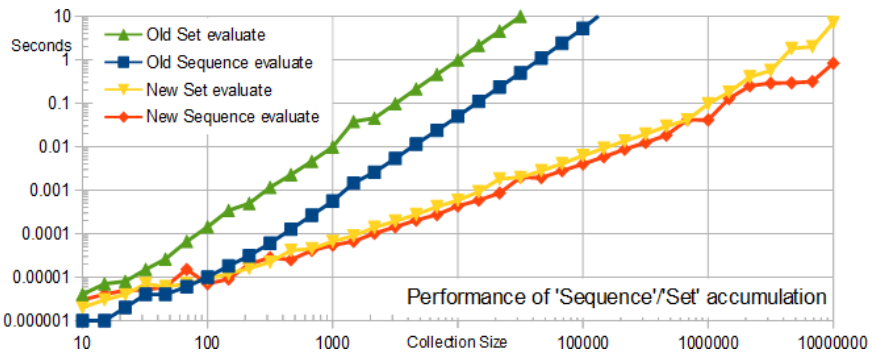

Fig. 7. 'Sequence' and 'Set' Accumulation Performance

4.7 Lazy limitations

Some operations such as `aCollection->size()` cannot be executed lazily since the size cannot be known without the whole collection. But in a suitable context such as `aCollection->size() > 3`, it is obvious that the full collection is not necessary after all. Even for `aCollection->size()`, `aCollection` does not need to be fully evaluated since we are only interested in the number of elements. If the computation of `aCollection` can be aware that only its size is required, a more efficient existence rather than value of each element might be computed.

4.8 Operation caches

As well as using 'lazy' evaluation to defer computation in the hope that it may prove redundant, performance may be improved by caching what has already been computed in the hope that it can be re-used.

As a side-effect free language, OCL is very well suited to caching the results of iteration or operation calls. However for simple arithmetic, short strings and small collections, the cost of caching and re-use may easily exceed the cost of re-computation. For larger collections, the cache size may be unattractive and the probability of re-use too low. Such dubious benefits perhaps explain the reticence of implementations to provide result caching.

Model to model transformations depend on re-use of created output elements and so the Eclipse QVTd tooling [8] pragmatically provides caches for Functions and Mappings but not Operations or Iterations.

Empirical observation suggests that for object operations and derived properties, the re-use benefits and statistics are much more favorable and so such caching should be part of an OCL evaluator. We will shortly see another example where operation caching can be helpful.

4.9 Smart select

The *select* iteration applies a Boolean predicate to filter a source collection.

`sourceCollection->select(booleanPredicate)`
In practice there are two common idioms associated with *select*.

Conformance selection: It is very common to use

```
S->select(oclIsKindOf(MyType)).oclAsType(MyType)
```

This selects those elements of `S` that conform to `MyType`. This clumsy test and cast idiom was recognized in OCL 2.4 and a `selectByKind()` operation added to improve readability.

In practice each source collection is partitioned into a very small number of types that can be identified by compile-time analysis of the OCL expressions. A naive implementation may recategorize the type of each element in each invocation. A more efficient implementation should re-use the type categorization to partition into all types of interest on the first invocation and cache the partitions for re-use by subsequent invocations for any of the types of interest. This should of course only be performed after Common Sub Expression or Loop Hoisting has eliminated redundant invocations, and only if there is more than one residual invocation.

Content selection: It is also common to use

```
S->select(element | element.name = wantedName)
```

This locates a matching content of `S` by choosing the appropriately named elements. This idiom treats the `S` as a `Map` with a `name` key, but whereas a `Map` returns the value in constant time, naive implementation of *select* incurs linear search cost.

For a single matching lookup, building the `Map` incurs a linear cost and so there is no benefit in an optimization. However in a larger application it is likely that the name lookup may occur a few times for the same name and many times for different names. Providing an underlying `Map` may be very beneficial, converting a quadratic performance to linear.

We will contrast the performance, with and without a `Map`, of the accumulation that computes

```
let S = Sequence{1..N} in
let t = S->collect(i|Tuple{x=i}) in
S->collect(i | t->select(x = i))
```

The first two *let* lines build the table all of whose entries are looked up by the final line.

The top line of Fig 8 shows the traditional naive full search for each lookup. The lower lines show the time to build the cache, the time to perform all lookups and their sum. The `Map` is clearly helpful for anything more than one lookup. As expected, it scales linearly rather than quadratically.

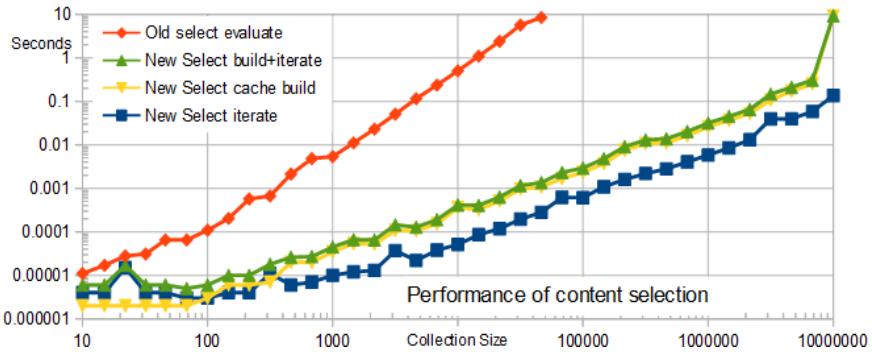

Fig. 8. 'select' Performance

5 Context and Status

The OCL tooling must perform OCL expression analyses to use the foregoing `NewCollection` capabilities effectively

- Identify mutable collections - use alternative mutable operation
- Identify single/multiple use collections - configure shared laziness
- Identify content selects - configure lookup tables

However since OCL by itself is useless, OCL tooling cannot know whether or how to optimize. It is only when OCL is embedded in a larger application that provides the models and the related OCL expressions that OCL becomes useful.

For the simplest OCL application in which an interactive OCL expression is evaluated with respect to a model, the costs of the model and expression analyses may easily outweigh the benefits. No optimization may well give the snappiest interactive response.

For a more complex OCL application such as the OCL definition of model constraints, operations and properties supported by OCLinEcore, Eclipse OCL provides a code generator [3] that embeds the Java for the OCL within the Java for the Ecore model.

The code generator performs a variety of compile-time analyses and syntheses:

- Common SubExpression / Loop hoisting
- Constant Folding
- Inlining
- Dispatch tables

The code generator also prepares tables and structures that cannot be fully analyzed until the actual run-time models are available:

– Run-Time Type Information (e.g. oclIsKindOf support)
– Run-Time Navigability Information (unnavigable opposites)
– Run-Time Instances Information (allInstances)

Adding a few additional activities is structurally easy, and only a minor compile-time degradation. The results presented earlier use a manual emulation of what the automated analysis and synthesis should achieve [2].

For OCL-based applications such as QVTc or QVTr [9], the Eclipse OCL code generator has been extended and appears to provide a twenty-fold speed-up compared to less optimized interpreted execution [4]. A smaller speed-up is to be expected for intensive *Collection* computations where most of the execution occurs in shared run-time support such as `Set::intersection()`.

6 Related Work

Lack of determinism in Model-to-Model transformation tools has been a regular irritation. e.g. https://bugs.eclipse.org/bugs/show_bug.cgi?id=358814.

Gogolla et al [1] identify the lack of determinism for OCL collection conversions and suggested that certain combinations should be deterministic so that the following is true:

```
SET->asBag()->asSequence() = SET->asSequence()
```

In this paper we make OCL collections fully deterministic and so all the suggested combinations are deterministic. The only open question is whether the deterministic order is *unknown*. If known, two different OCL implementations should yield the same deterministic result.

Lazy OCL evaluation is used by Tisi et al [2] to support infinite collections. The authors consider their work as a variant semantics for OCL. Our alternative reading of the OCL specification allows infinite collections to be supported by regular OCL tooling provided eager operations such as `Collection::size()` are avoided. The default `Bag`-aware iteration provided by the `NewCollection` is incompatible with lazy Bags, however an alternative but less efficient approach could remedy this limitation.

Discomfort with the prevailing state of the art highlighted by these papers inspired the solution provided in this paper. The unified Collection implementation type is new. The deterministic Collection type is new. 'Lazy' OCL is not new, but the OCL expression analysis to exploit the lazy unified Collection type is new.

[2] Unifying the four concrete eager Collection types by a single lazy replacement is an API breakage that requires Eclipse OCL to make a major version number change. The code for lazy evaluations is therefore only available on the ewillink/509670 branch in the Eclipse OCL GIT repository

7 Conclusions

We have introduced a new underlying representation for a Collection implementation that unifies all four types and eliminates redundant conversion costs.

The new representation is deterministic allowing OCL and OCL-based model-to-model transformation tools to be deterministic too.

We have distinguished between program and machine failures so that the new representation can provide effective lazy evaluation capabilities.

We have used lazy evaluation to significantly reduce memory costs and to avoid redundant computations by allowing favorable algorithms to terminate prematurely.

We have linearized some quadratic costs by using mutable collections and a content cache for select().

References

1. Gogolla, M., Hilken, F.: Making OCL Collection Operations More Deterministic with Restricting Equations, 16th International Workshop in OCL and Textual Modeling, October 2, 2016, Saint-Malo, France. http://www.db.informatik.uni-bremen.de/publications/intern/ocl2016-talk-lightning-mg-fh.pdf
2. Tisi, M., Douence, R., Wagelaar, D.: Lazy Evaluation for OCL. 15th International Workshop on OCL and Textual Modeling, September 8, 2015, Ottawa, Canada. https://ocl2015.lri.fr/OCL_2015_paper_1111_1115.pdf
3. Willink, E.: An extensible OCL Virtual Machine and Code Generator. 2012 Workshop on OCL and Textual Modelling (OCL 2012), September 30, 2012, Innsbruck, Austria. http://st.inf.tu-dresden.de/OCL2012/preproceedings/14.pdf
4. Willink, E: Local Optimizations in Eclipse QVTc and QVTr using the Micro-Mapping Model of Computation, 2nd International Workshop on Executable Modeling, Exe 2016, Saint-Malo, October 2016. http://eclipse.org/mmt/qvt/docs/EXE2016/MicroMappings.pdf
5. Willink, E.: Safe Navigation in OCL. 15th International Workshop on OCL and Textual Modeling, September 8, 2015, Ottawa, Canada. https://ocl2015.lri.fr/OCL_2015_paper_1111_1400.pdf
6. Dresden OCL Project. http://www.dresden-ocl.org/index.php/DresdenOCL
7. Eclipse OCL Project. https://projects.eclipse.org/projects/modeling.mdt.ocl
8. Eclipse QVT Declarative Project. https://projects.eclipse.org/projects/modeling.mmt.qvtd
9. OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.3. OMG Document Number: ptc/16-06-03, June 2016.
10. OMG Unified Modeling Language (OMG UML), Version 2.5, OMG Document Number: formal/15-03-01, Object Management Group (2015), http://www.omg.org/spec/UML/2.5
11. Object Constraint Language. Version 2.4., OMG Document Number: formal/2014-02-03, Object Management Group (2009), http://www.omg.org/spec/OCL/2.4
12. USE, The UML-based Specification Environment. http://useocl.sourceforge.net/w/index.php/Main_Page