Dynamically Composing Collection Operations through Collection Promises

Juan Pablo Sandoval Alcocer, Marcus Denker, Alexandre Bergel, Yasett Acurana

Last November in Chile...

Discussing with Juan Pablo about his research

[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.

A considerable number of performance bugs and regressions are related with loops involving collections.

Problem

- Filtering, mapping, and iterating collections are frequent operations in Smalltalk
- It create **lots** of intermediate collections

Example

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

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

Properties

- Cross method boundaries
- Might even be stored in a variable for readability

Current solutions (1)

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

- Lots of these defined in Pharo
- Only possible inside one method
- Code needs to be rewritten

Current solutions (2)

- We could use a stream based iteration protocol
- Code needs to be rewritten

- Not as easily composable
  - Will be useful, but not for all cases

Collection Promises

- Delay operations, merge later
- Simple prototype to evaluate if this idea makes sense

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

```
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 }.
```

- handle select: & similar:

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

- all others: DNU handler

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

Performance: simple bench

- **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**, using the method select:thenCollect: directly.

Performance: result

- Run for different Collection sizes
- Result:
  - Slower than rewrite
  - Faster than creating intermediate collection
  - Collection size matters: better with large collections.

Details: see Paper

Result (for us)

- We wanted to know: does it make sense?
- Very simple prototype shows that it is promising
  - Even though very simple implementation

- Result: Yes, we should continue

Future Work

- Extend to cover more cases
- Can we automatically detect where intermediate collections are created?
- Can we detect hotspots?
- Can we reflectively introduce promises?
- Try to see if we can get speed-up in practice

Questions ?