Storage Strategies for Collections in Dynamically Typed Languages

Carl Friedrich Bolz
Lukas Diekmann
Laurence Tratt

King's College London
Software Development Team
2013-08-24
Collections in dynamically typed languages are **slow**
Why is that?
Referencing arbitrary types
Referencing arbitrary types

Needs common representation on heap
Memory usage in dynamically typed languages

Referencing arbitrary types

Needs **common representation** on heap

Solution: **Boxing**
Example

Statically typed integer vs dynamically typed integer

<table>
<thead>
<tr>
<th>Language</th>
<th>Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>PyPy</td>
<td></td>
</tr>
</tbody>
</table>
Statically typed integer vs dynamically typed integer

<table>
<thead>
<tr>
<th>Language</th>
<th>Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>PyPy</td>
<td></td>
</tr>
</tbody>
</table>
Statically typed integer vs dynamically typed integer

<table>
<thead>
<tr>
<th>Language</th>
<th>Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>PyPy</td>
<td>3</td>
</tr>
</tbody>
</table>
Worse in collections

```python
list = [1, 2, 3]
```

>>> list = [1, 2, 3]
Worse in collections

A list with 1,000,000 integers

<table>
<thead>
<tr>
<th>Language</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>PyPy</td>
<td></td>
</tr>
</tbody>
</table>
Worse in collections

A list with 1,000,000 integers

<table>
<thead>
<tr>
<th>Language</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>3.8 MiB</td>
</tr>
<tr>
<td>PyPy</td>
<td></td>
</tr>
</tbody>
</table>
A list with 1,000,000 integers

<table>
<thead>
<tr>
<th>Language</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>3.8 MiB</td>
</tr>
<tr>
<td>PyPy</td>
<td>15.3 MiB</td>
</tr>
</tbody>
</table>
• **H1** It is *uncommon* for a homogeneously typed collection storing objects of primitive types to later type dehomogenise.

• **H2** When a previously homogeneously typed collection type dehomogenises, the transition happens after only a *small number of elements* have been added.
A way out?

- **H1** It is uncommon for a homogeneously typed collection storing objects of primitive types to later type dehomogenise.

- **H2** When a previously homogeneously typed collection type dehomogenises, the transition happens after only a small number of elements have been added.
• **H1** It is uncommon for a homogeneously typed collection storing objects of primitive types to later type dehomogenise.

• **H2** When a previously homogeneously typed collection type dehomogenises, the transition happens after only a small number of elements have been added.
A way out?

And now?
And now?

Storage strategies
Split up collection into strategy and storage
Strategy
Takes over collections functionality
Strategy
Takes over collections functionality

Storage
Contains *unboxed* data types
Storage strategies

\[
11 = [1, 2, 3] \\
12 = [4, 5, 6]
\]
Storage strategy life cycle

- EmptyStrategy
  - Add integer
  - Add object
  - Empty

- IntegerStrategy
  - Add integer
  - Add object

- ObjectStrategy
  - Add object

- Strategy
  - Add ...

http://soft-dev.org/
class W_ListObject(W_Object):
    def __init__(self):
        self.strategy = EmptyListStrategy()
        self.lstorage = None

    def append(self, w_item):
        self.strategy.append(self, w_item)

@singleton
class ListStrategy(object):
    def append(self, w_list, w_item):
        raise NotImplementedError("abstract")

@singleton
class EmptyListStrategy(ListStrategy):
    def append(self, w_list, w_item):
        if is_boxed_int(w_item):
            w_list.strategy = IntegerListStrategy()
            w_list.lstorage = new_empty_int_list()
        elif ...
        else:
            w_list.strategy = ObjectListStrategy()
            w_list.lstorage = new_empty_object_list()
        w_list.append(w_item)

@singleton
class IntegerListStrategy(ListStrategy):
    def append(self, self, w_list, w_item):
        if is_boxed_int(w_item):
            u_int = unbox_int(w_item)
            w_list.lstorage.append_int(u_int)
        return self.switch_to_object_strategy(w_list)
        w_list.append(w_item)

    def switch_to_object_strategy(self, w_list):
        lstorage = new_empty_object_list()
        for i in w_list.lstorage:
            lstorage.append_obj(box_int(i))
        w_list.strategy = ObjectListStrategy()
        w_list.lstorage = lstorage

@singleton
class ObjectListStrategy(ListStrategy):
    def append(self, self, w_list, w_item):
        w_list.lstorage.append_obj(w_item)
Unboxed data types enable several optimisations.
Start with a predefined strategy.

"string".split("i")
Copy strategy from other collection

```
set([1,2,3])
```
Add fast paths when comparing objects of different types.

contains, difference, issubset
RangeListStrategy: Calculates elements on the fly.

```
range(1000)
```
unboxed data types
JIT interaction

unboxed data types

less type checks in traces
JIT interaction

unboxed data types

- less type checks in traces
- faster comparisons
Evaluation
Methodology

• Classic performance benchmarks
• Real world applications
• Experiments downloadable at soft-dev.org
• **H1** It is *uncommon* for a homogeneously typed collection storing objects of primitive types to later type dehomogenise.

• **H2** When a previously homogeneously typed collection type dehomogenises, the transition happens after only a *small number of elements* have been added.
List transitions

- EmptyListStrategy
- FloatListStrategy
- IntegerListStrategy
- StringListStrategy
- RangeListStrategy
- ObjectListStrategy
Validating hypotheses

- **H1** It is uncommon for a homogeneously typed collection storing objects of primitive types to later type dehomogenise.

- **H2** When a previously homogeneously typed collection type dehomogenises, the transition happens after only a small number of elements have been added.
Validating hypotheses

- **H1** It is uncommon for a homogeneously typed collection storing objects of primitive types to later type dehomogenise.

- **H2** When a previously homogeneously typed collection type dehomogenises, the transition happens after only a small number of elements have been added.
Validating hypotheses

- **H1** It is uncommon for a homogeneously typed collection storing objects of primitive types to later type dehomogenise.

- **H2** When a previously homogeneously typed collection type dehomogenises, the transition happens after only a small number of elements have been added.
• Executed 35 times
• Skip first 5 results containing JIT warmup
Speed benchmarks

![Graph showing relative execution time for various software tools.](http://soft-dev.org/)
Performance increase of $\sim 18\%$
Memory benchmarks

- Memory usage via probes
- At probably maximum point of maximal usage
Memory benchmarks

Relative memory usage

<table>
<thead>
<tr>
<th>Library</th>
<th>Relative Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>disaster</td>
<td>1</td>
</tr>
<tr>
<td>feedparser</td>
<td>0.96</td>
</tr>
<tr>
<td>invindex</td>
<td>0.84</td>
</tr>
<tr>
<td>multiwords</td>
<td>0.72</td>
</tr>
<tr>
<td>networkx</td>
<td>0.60</td>
</tr>
<tr>
<td>nltk-wordassoc</td>
<td>0.50</td>
</tr>
<tr>
<td>orm</td>
<td>0.40</td>
</tr>
<tr>
<td>pydblite</td>
<td>0.30</td>
</tr>
<tr>
<td>pyexcelerator</td>
<td>0.20</td>
</tr>
<tr>
<td>scapy</td>
<td>0.10</td>
</tr>
<tr>
<td>slowsets</td>
<td>0.05</td>
</tr>
<tr>
<td>whoosh</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Memory decrease of $\sim 6\%$
Memory benchmarks

Memory decrease of \( \sim 6\% \)
Summary

• Simple to implement

• Only ∼1500 LoC

• Good performance boost: ∼18%

• Minor memory reduction: ∼6%
Summary

• simple to implement
Summary

- simple to implement
- only \( \sim 1500 \) LoC
Summary

- simple to implement
- only \( \sim 1500 \) LoC
- good performance boost: \( \sim 18\% \)
Summary

- simple to implement
- only \(~1500\) LoC
- good performance boost: \(~18\%\)
- minor memory reduction: \(~6\%\)
Thank you for listening

http://soft-dev.org/