Value and Allocation Sensitivity in Static Python Analyses

Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné

https://rmonat.fr/soap20/
Introduction
Python, from a PL perspective

- #2 language on Github,
- Object oriented,
- Dynamic typing: types are only known at runtime,
- Allows operator redefinition for custom classes,
- Introspection,
- Self-modification,
- `eval`.

Python, from a PL perspective

- #2 language on Github,
- Object oriented,
Python, from a PL perspective

- #2 language on Github,
- Object oriented,
- Dynamic typing: types are only known at runtime,
# Python, from a PL perspective

- #2 language on Github,
- Object oriented,
- Dynamic typing: types are only known at runtime,
- Allows operator redefinition for custom classes,
Python, from a PL perspective

- #2 language on Github,
- Object oriented,
- Dynamic typing: types are only known at runtime,
- Allows operator redefinition for custom classes,
- Introspection,
Python, from a PL perspective

- #2 language on Github,
- Object oriented,
- Dynamic typing: types are only known at runtime,
- Allows operator redefinition for custom classes,
- Introspection,
- Self-modification,
Python, from a PL perspective

- #2 language on Github,
- Object oriented,
- Dynamic typing: types are only known at runtime,
- Allows operator redefinition for custom classes,
- Introspection,
- Self-modification,
- eval.
Static Analysis of Python

Based on a previous work focusing on sound type analysis\textsuperscript{1}.

\textsuperscript{1}Monat, Ouadjaout, and Miné. “Static Type Analysis by Abstract Interpretation of Python Programs”. ECOOP’20.
\textsuperscript{2}Journault et al. “Combinations of reusable abstract domains for a multilingual static analyzer”. VSTTE’19.
Based on a previous work focusing on sound type analysis\textsuperscript{1}.

**Goal:** detect all potential runtime errors (i.e., uncaught exceptions).

\textsuperscript{1}Monat, Ouadjaout, and Miné. “Static Type Analysis by Abstract Interpretation of Python Programs”. ECOOP’20.

\textsuperscript{2}Journault et al. “Combinations of reusable abstract domains for a multilingual static analyzer”. VSTTE’19.
Based on a previous work focusing on sound type analysis\textsuperscript{1}.

**Goal:** detect all potential runtime errors (i.e, uncaught exceptions).

**Difficulties:**

- Sound approximation of the semantics.
- Supporting a decent language & library subset.

---

\textsuperscript{1} Monat, Ouadjaout, and Miné. “Static Type Analysis by Abstract Interpretation of Python Programs”. ECOOP’20.

\textsuperscript{2} Journault et al. “Combinations of reusable abstract domains for a multilingual static analyzer”. VSTTE’19.
Static Analysis of Python

Based on a previous work focusing on sound type analysis\(^1\).

**Goal:** detect all potential runtime errors (i.e., uncaught exceptions).

**Difficulties:**
- Sound approximation of the semantics.
- Supporting a decent language & library subset.

**Implementation:** Mopsa\(^2\), easing static analysis development through modular, reusable domains (ex: shared domains for C and Python).

---

\(^1\) Monat, Ouadjaout, and Miné. “Static Type Analysis by Abstract Interpretation of Python Programs”. ECOOP’20.

Studied Research Questions

▶ Type or Value Analysis?
  ● Cost of adding value sensitivity?
  ● Precision gain?

▶ Heap Abstraction using the Recency Abstraction
  ● Best policy to group/abstract addresses?
  ● Policy depending on the value-sensitivity?

▶ Abstract Garbage Collection
  ● Is the gain provided sufficient to offset its cost?

3Balakrishnan and Reps. “Recency-Abstraction for Heap-Allocated Storage”. SAS.
Studied Research Questions

- **Type or Value Analysis?**
  - Cost of adding value sensitivity?
  - Precision gain?

- **Heap Abstraction using the Recency Abstraction**
  - Best policy to group/abstract addresses?
  - Policy depending on the value-sensitivity?

- **Abstract Garbage Collection**
  - Is the gain provided sufficient to offset its cost?

---

3Balakrishnan and Reps. “Recency-Abstraction for Heap-Allocated Storage”. SAS.
Studied Research Questions

- Type or Value Analysis?
  - Cost of adding value sensitivity?
  - Precision gain?

- Heap Abstraction using the Recency Abstraction\(^3\)
  - Best policy to group/abstract addresses?
  - Policy depending on the value-sensitivity?

- Abstract Garbage Collection
  - Is the gain provided sufficient to offset its cost?

\(^3\)Balakrishnan and Reps. “Recency-Abstraction for Heap-Allocated Storage”. SAS.
Type and Value Analyses of Python
Type Analysis

- By induction on the syntax
- Context-sensitive
- Flow-sensitive
By induction on the syntax

Context-sensitive

Flow-sensitive

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(3), Task(5)]
m = 0
for i in range(len(l)):
    m = m + l[i].weight
m = m // (i + 1)
```
Type Analysis

- By induction on the syntax
- Context-sensitive
- Flow-sensitive

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(3), Task(5)]
m = 0
for i in range(len(l)):
    m = m + l[i].weight
m = m // (i + 1)
```
Type Analysis

- By induction on the syntax
- Context-sensitive
- Flow-sensitive

```
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(3), Task(5)]
m = 0
for i in range(len(l)):
    m = m + l[i].weight
m = m // (i + 1)
```

1 ValueError
Type Analysis

- By induction on the syntax
- Context-sensitive
- Flow-sensitive

1 class Task:
2     def __init__(self, weight):
3         if weight < 0: raise ValueError
4         self.weight = weight

5 l = [Task(2), Task(1), Task(3), Task(5)]
6 m = 0
7 for i in range(len(l)):
8     m = m + l[i].weight
9     m = m // (i + 1)

▶ 1+3 ValueErrors
Type Analysis

▶ By induction on the syntax
▶ Context-sensitive
▶ Flow-sensitive

1. **class** Task:
2.   def __init__(self, weight):
3.     if weight < 0: raise ValueError
4.     self.weight = weight

5. l = [Task(2), Task(1), Task(3), Task(5)]
6. m = 0
7. for i in range(len(l)):
8.   m = m + l[i].weight
9. m = m // (i + 1)

▶ 4 ValueError
▶ l: list of Tasks, having an integer weight

- **IndexError** (over i)
- **NameError**
- **ZeroDivisionError** =⇒ 7 false alarms
Type Analysis

- By induction on the syntax
- Context-sensitive
- Flow-sensitive

```
1 class Task:
2     def __init__(self, weight):
3         if weight < 0: raise ValueError
4         self.weight = weight
5
6     l = [Task(2), Task(1), Task(3), Task(5)]
7     m = 0
8     for i in range(len(l)):
9         m = m + l[i].weight
10        m = m // (i + 1)
```

• 4 ValueErrors
• l: list of Tasks, having an integer weight
• i, m: integers
Type Analysis

▶ By induction on the syntax
▶ Context-sensitive
▶ Flow-sensitive

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(3), Task(5)]
m = 0
for i in range(len(l)):
    m = m + l[i].weight
m = m // (i + 1)
```

▶ 4 ValueErrors
▶ l: list of Tasks, having an integer weight
▶ i, m: integers
▶ IndexError

IndexError
Type Analysis

- By induction on the syntax
- Context-sensitive
- Flow-sensitive

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(3), Task(5)]
m = 0
for i in range(len(l)):
    m = m + l[i].weight
m = m // (i + 1)
```

- 4 ValueErrors
- l: list of Tasks, having an integer weight
- i, m: integers
- IndexError
- NameError (over i)
- ZeroDivisionError

⇒ 7 false alarms
Refinement of the type analysis.

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(3), Task(5)]

m = 0
for i in range(len(l)):
    m = m + l[i].weight
m = m // (i + 1)
```
Value Analysis

Refinement of the type analysis.

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(3), Task(5)]
m = 0
for i in range(len(l)):
    m = m + l[i].weight
m = m // (i + 1)
```
Value Analysis

Reinfnement of the type analysis.

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(3), Task(5)]
m = 0
for i in range(len(l)):
    m = m + l[i].weight
m = m // (i + 1)
```

- l: list of Tasks, having an integer weight; $1 \leq \text{weight} \leq 5$
- len(l) = 4
Value Analysis

Refinement of the type analysis.

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(3), Task(5)]

m = 0
for i in range(len(l)):
    m = m + l[i].weight
m = m // (i + 1)
```

- l: list of `Tasks`, having an integer weight; $1 \leq \text{weight} \leq 5$
- len(l) = 4
- $0 \leq i < 4$
Value Analysis

Refinement of the type analysis.

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(3), Task(5)]
m = 0
for i in range(len(l)):
    m = m + l[i].weight
m = m // (i + 1)
```

- l: list of Tasks, having an integer weight; $1 \leq weight \leq 5$
- len(l) = 4
- $0 \leq i < 4$
- valid list access
Refinement of the type analysis.

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(3), Task(5)]
m = 0
for i in range(len(l)):
    m = m + l[i].weight
m = m // (i + 1)
```

- \( l \): list of \texttt{Tasks}, having an integer weight; \( 1 \leq \text{weight} \leq 5 \)
- \( \text{len}(l) = 4 \)
- \( i = 3 \)
Refinement of the type analysis.

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(3), Task(5)]

m = 0
for i in range(len(l)):
    m = m + l[i].weight
m = m // (i + 1)

▶ l: list of Tasks, having an integer weight; 1 ≤ weight ≤ 5
▶ len(l) = 4
▶ i = 3
⇒⇒ No false alarm!
```
Related Work: Static Analyses of Python

▶ Type Analyses:
  - Dataflow analysis by Fritz and Hage[^4].
  - SMT-based type inference[^5].
  - Pytype, a tool used by Google[^6].

▶ Value Analysis by Fromherz et al[^7].

[^4]: Fritz and Hage. “Cost versus precision for approximate typing for Python”. PEPM.
[^5]: Hassan et al. “MaxSMT-Based Type Inference for Python 3”. CAV.
[^6]: Kramm et al. Pytype.
[^7]: Fromherz, Ouadjaout, and Miné. “Static Value Analysis of Python Programs by Abstract Interpretation”. NFM.
### Experimental Evaluation

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Type Analysis</th>
<th>Value Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time</td>
<td>Mem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Type</td>
<td>Index</td>
</tr>
<tr>
<td>scimark</td>
<td>416</td>
<td>1.4s</td>
<td>12MB</td>
</tr>
<tr>
<td>richards</td>
<td>426</td>
<td>13s</td>
<td>112MB</td>
</tr>
<tr>
<td>unpack</td>
<td>458</td>
<td>8.3s</td>
<td>7MB</td>
</tr>
<tr>
<td>go</td>
<td>461</td>
<td>27s</td>
<td>345MB</td>
</tr>
<tr>
<td>hexiom</td>
<td>674</td>
<td>1.1m</td>
<td>525MB</td>
</tr>
<tr>
<td>regex</td>
<td>1792</td>
<td>23s</td>
<td>18MB</td>
</tr>
<tr>
<td>process</td>
<td>1417</td>
<td>10s</td>
<td>64MB</td>
</tr>
<tr>
<td>choose</td>
<td>2562</td>
<td>1.1m</td>
<td>1.6GB</td>
</tr>
<tr>
<td>Total</td>
<td>9294</td>
<td>4.0m</td>
<td>2.8GB</td>
</tr>
</tbody>
</table>
## Experimental Evaluation

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Type Analysis</th>
<th>Value Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time</td>
<td>Mem.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mem.</td>
<td></td>
</tr>
<tr>
<td>scimark</td>
<td>416</td>
<td>1.4s</td>
<td>12MB</td>
</tr>
<tr>
<td>richards</td>
<td>426</td>
<td>13s</td>
<td>112MB</td>
</tr>
<tr>
<td>unpack</td>
<td>458</td>
<td>8.3s</td>
<td>7MB</td>
</tr>
<tr>
<td>go</td>
<td>461</td>
<td>27s</td>
<td></td>
</tr>
<tr>
<td>hexiom</td>
<td>674</td>
<td>1.1m</td>
<td></td>
</tr>
<tr>
<td>regex</td>
<td>1792</td>
<td>23s</td>
<td></td>
</tr>
<tr>
<td>process</td>
<td>1417</td>
<td>10s</td>
<td></td>
</tr>
<tr>
<td>choose</td>
<td>2562</td>
<td>1.1m</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>9294</td>
<td>4.0m</td>
<td>2.8GB</td>
</tr>
</tbody>
</table>

### Conclusion

The Value Analysis:
- does not remove false type alarms,
- significantly reduces index errors,
- is $\approx 3 \times$ costlier.
Variable-Policy Recency Abstraction
Presenting the Recency Abstraction

Goal: Allow a precise analysis of object initialization, assuming it happens just after allocation.
Presenting the Recency Abstraction

**Goal:** Allow a precise analysis of object initialization, assuming it happens just after allocation.

**Means:** Twofold partitioning:

1. by allocation site \( l \in \mathbb{L} \)
2. through a recency criterion
   - \((l, r)\): most recent allocation (with strong updates)
   - \((l, o)\): older addresses (summarized)
Presenting the Recency Abstraction

Goal: Allow a precise analysis of object initialization, assuming it happens just after allocation.

Means: Twofold partitioning:

1. by allocation site \( l \in \mathbb{L} \)
2. through a recency criterion
   - \((l, r)\): most recent allocation (with strong updates)
   - \((l, o)\): older addresses (summarized)

First try: type analysis \( \rightsquigarrow \)
replace \( l \in \mathbb{L} \) with Python types \( t \in \mathbb{T} \).
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(4), Task(5)]
Presenting the Recency Abstraction

class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(4), Task(5)]

Allocation:
Presenting the Recency Abstraction

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(4), Task(5)]
```

Initialization:

\[
\{(\text{Task}, r) \cdot \text{weight} \mapsto [2, 2]\}
\]
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(4), Task(5)]

Allocation: (Task, r) \sim (Task, o)

\{ (Task, r) \cdot weight \mapsto [2, 2] \\
\{ (Task, o) \cdot weight \mapsto [2, 2] \\

Presenting the Recency Abstraction

```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(4), Task(5)]
```

Initialization:

\[
\begin{aligned}
\{(\text{Task}, r) \cdot \text{weight} \mapsto [2, 2]\} \\
\{(\text{Task}, r) \cdot \text{weight} \mapsto [1, 1]\} \\
\{(\text{Task}, o) \cdot \text{weight} \mapsto [2, 2]\}
\end{aligned}
\]
Presenting the Recency Abstraction

class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(4), Task(5)]

Allocation: (Task, r) ↦ (Task, o)

\[
\begin{cases}
  (\text{Task}, r) \cdot \text{weight} \mapsto [2, 2] \\
  (\text{Task}, r) \cdot \text{weight} \mapsto [1, 1] \\
  (\text{Task}, o) \cdot \text{weight} \mapsto [2, 2] \\
  (\text{Task}, r) \quad (\text{Task}, o) \cdot \text{weight} \mapsto [2, 2] \sqcup [1, 1]
\end{cases}
\]
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(4), Task(5)]

Initialization:

\[
\begin{align*}
    (\text{Task}, r) \cdot \text{weight} & \mapsto [2, 2] \\
    (\text{Task}, o) \cdot \text{weight} & \mapsto [1, 1] \\
    (\text{Task}, r) \cdot \text{weight} & \mapsto [4, 4] \\
    (\text{Task}, o) \cdot \text{weight} & \mapsto [1, 2]
\end{align*}
\]
```python
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(4), Task(5)]

Allocation: (Task, r) \mapsto (Task, o)
```

\[
\begin{align*}
\{(Task, r) \cdot weight \mapsto [2, 2] \\
\{(Task, r) \cdot weight \mapsto [1, 1] \\
\{(Task, o) \cdot weight \mapsto [2, 2] \\
\{(Task, r) \cdot weight \mapsto [4, 4] \\
\{(Task, o) \cdot weight \mapsto [1, 2] \\
\end{align*}
\]
class Task:
    def __init__(self, weight):
        if weight < 0: raise ValueError
        self.weight = weight

l = [Task(2), Task(1), Task(4), Task(5)]

Initialization:

{(Task, r) \cdot weight \mapsto [2, 2]}
{(Task, r) \cdot weight \mapsto [1, 1]}
{(Task, o) \cdot weight \mapsto [2, 2]}
{(Task, r) \cdot weight \mapsto [4, 4]}
{(Task, o) \cdot weight \mapsto [1, 2]}
{(Task, r) \cdot weight \mapsto [5, 5]}
{(Task, o) \cdot weight \mapsto [1, 4]}
The type-based partitioning may be unprecise in some cases:

```
for i in range(L):
    for j in range(M):
        for k in range(N):
            ...
```
Variable Allocation Policies

The type-based partitioning may be unprecise in some cases:

```
1    for i in range(L):
2        for j in range(M):
3            for k in range(N):
4                ...
```

`range(L), range(M)` summarized in `(range, o) \implies` mixed ranges for i and j!
The type-based partitioning may be unprecise in some cases:

```
1  for i in range(L):
2      for j in range(M):
3          for k in range(N):
4          ...
```

`range(L), range(M)` summarized in `(range, o) ➞ mixed ranges for i and j!`

**Variable Allocation Policies:** type-based and location-based partitioning. Parameterized by the user given the type. Details in the paper.
## Experimental Evaluation

<table>
<thead>
<tr>
<th>Name</th>
<th>Type Only Partitioning</th>
<th></th>
<th></th>
<th>Type + Loc. Partitioning</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Mem.</td>
<td></td>
<td>Time</td>
<td>Mem.</td>
<td></td>
</tr>
<tr>
<td>chaoss</td>
<td>30s</td>
<td>197MB</td>
<td>30</td>
<td>2.4m</td>
<td>1.2GB</td>
<td>15</td>
</tr>
<tr>
<td>rayt</td>
<td>27s</td>
<td>171MB</td>
<td>28</td>
<td>4.5s</td>
<td>74MB</td>
<td>7</td>
</tr>
<tr>
<td>scimark</td>
<td>3.4s</td>
<td>27MB</td>
<td>4</td>
<td>3.0s</td>
<td>27MB</td>
<td>3</td>
</tr>
<tr>
<td>richards</td>
<td>17s</td>
<td>149MB</td>
<td>5</td>
<td>69m</td>
<td>15GB</td>
<td>5</td>
</tr>
<tr>
<td>unpack</td>
<td>9.4s</td>
<td>6MB</td>
<td>0</td>
<td>9.6s</td>
<td>6MB</td>
<td>0</td>
</tr>
<tr>
<td>go</td>
<td>2.0m</td>
<td>1.4GB</td>
<td>57</td>
<td>1.7m</td>
<td>1.2GB</td>
<td>57</td>
</tr>
<tr>
<td>hexiom</td>
<td>4.7m</td>
<td>3.2GB</td>
<td>27</td>
<td>4.2m</td>
<td>3.2GB</td>
<td>27</td>
</tr>
<tr>
<td>regex</td>
<td>1.3m</td>
<td>56MB</td>
<td>145</td>
<td>3.6m</td>
<td>85MB</td>
<td>145</td>
</tr>
<tr>
<td>process</td>
<td>12s</td>
<td>85MB</td>
<td>15</td>
<td>11s</td>
<td>74MB</td>
<td>13</td>
</tr>
<tr>
<td>choose</td>
<td>2.9m</td>
<td>3.7GB</td>
<td>68</td>
<td>3.1m</td>
<td>4.3GB</td>
<td>63</td>
</tr>
<tr>
<td>Total</td>
<td>13m</td>
<td>9.1GB</td>
<td>392</td>
<td>87m</td>
<td>25GB</td>
<td>359</td>
</tr>
</tbody>
</table>
## Experimental Evaluation

<table>
<thead>
<tr>
<th>Name</th>
<th>Type Only Partitioning</th>
<th></th>
<th>Type + Loc. Partitioning</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Mem.</td>
<td></td>
<td>Time</td>
</tr>
<tr>
<td>chaos</td>
<td>30s</td>
<td>197MB</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>rayt</td>
<td>27s</td>
<td>171MB</td>
<td></td>
<td>4.5s</td>
</tr>
<tr>
<td>scimark</td>
<td>3.4s</td>
<td>27MB</td>
<td></td>
<td>3.0s</td>
</tr>
<tr>
<td>richards</td>
<td>17s</td>
<td>149MB</td>
<td></td>
<td>3.6m</td>
</tr>
<tr>
<td>unpack</td>
<td>9.4s</td>
<td>6MB</td>
<td>0</td>
<td>9.6s</td>
</tr>
<tr>
<td>go</td>
<td>2.0m</td>
<td>1.4GB</td>
<td></td>
<td>1.7m</td>
</tr>
<tr>
<td>hexiom</td>
<td>4.7m</td>
<td>3.2GB</td>
<td></td>
<td>4.2m</td>
</tr>
<tr>
<td>regex</td>
<td>1.3m</td>
<td>56MB</td>
<td>145</td>
<td>3.6m</td>
</tr>
<tr>
<td>process</td>
<td>12s</td>
<td>85MB</td>
<td>15</td>
<td>11s</td>
</tr>
<tr>
<td>choose</td>
<td>2.9m</td>
<td>3.7GB</td>
<td>68</td>
<td>3.1m</td>
</tr>
<tr>
<td>Total</td>
<td>13m</td>
<td>9.1GB</td>
<td>392</td>
<td>87m</td>
</tr>
</tbody>
</table>

**Conclusion**

In most cases:
- Type-only partitioning is more efficient
- But it induces a 9% alarm overhead
Abstract Garbage Collection
Previously, all allocated abstract addresses were kept in the state.
Previously, all allocated abstract addresses were kept in the state.

```python
1  def f():
2      l = [Task(2), Task(1), Task(4), Task(5)]
3      return sum([x.weight for x in l])
4
5  s = f()
6  [...]

(Task, r), (Task, o) still exist after line 6.
We implemented a tracing AGC, reducing abstract state’s size.
Abstract Garbage Collection

Previously, all allocated abstract addresses were kept in the state.

```python
1   def f():
2       l = [Task(2), Task(1), Task(4), Task(5)]
3       return sum([x.weight for x in l])

4

5   s = f()
6   [...]
```

(Task, r), (Task, o) still exist after line 6.
We implemented a tracing AGC, reducing abstract state’s size.

AGC may improve precision (example in the paper); it does not happen in our benchmarks though.
Related Work

- Analysis of higher-order languages: tracing AGC\(^8\), reference-counting AGC\(^9\). Reduces analysis time by an order of magnitude. May improve the precision.
- Analysis of object-oriented languages: TAJS\(^10\): reduces memory.

---


\(^9\)Es, Stiévenart, and Roover. “Garbage-Free Abstract Interpretation Through Abstract Reference Counting”. ECOOP.

\(^10\)Jensen, Møller, and Thiemann. “Type Analysis for JavaScript”. SAS.
## Experimental Evaluation

<table>
<thead>
<tr>
<th>Name</th>
<th>Without AGC</th>
<th>With AGC</th>
<th>Rel. Improv.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time Mem.</td>
<td>Time Mem.</td>
<td>Time Mem.</td>
</tr>
<tr>
<td>chaos</td>
<td>10s 64MB</td>
<td>7.4s 42MB</td>
<td>28% 34%</td>
</tr>
<tr>
<td>rayt</td>
<td>17s 74MB</td>
<td>14s 74MB</td>
<td>16% 0%</td>
</tr>
<tr>
<td>scimark</td>
<td>1.5s 13MB</td>
<td>1.4s 12MB</td>
<td>5% 8%</td>
</tr>
<tr>
<td>richards</td>
<td>16s 227MB</td>
<td>13s 112MB</td>
<td>21% 51%</td>
</tr>
<tr>
<td>unpack</td>
<td>10s 9MB</td>
<td>8.3s 7MB</td>
<td>19% 22%</td>
</tr>
<tr>
<td>go</td>
<td>38s 604MB</td>
<td>27s 345MB</td>
<td>31% 43%</td>
</tr>
<tr>
<td>hexiom</td>
<td>2.2m 1.1GB</td>
<td>1.1m 525MB</td>
<td>49% 50%</td>
</tr>
<tr>
<td>regex</td>
<td>30s 24MB</td>
<td>23s 18MB</td>
<td>23% 25%</td>
</tr>
<tr>
<td>process</td>
<td>14s 85MB</td>
<td>10s 64MB</td>
<td>28% 25%</td>
</tr>
<tr>
<td>choose</td>
<td>2.0m 3.2GB</td>
<td>1.1m 1.6GB</td>
<td>43% 50%</td>
</tr>
<tr>
<td>Total</td>
<td>6.5m 5.4GB</td>
<td>4.0m 2.8GB</td>
<td>38% 47%</td>
</tr>
</tbody>
</table>
Experimental Evaluation

<table>
<thead>
<tr>
<th>Name</th>
<th>Without AGC</th>
<th>With AGC</th>
<th>Rel. Impr.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Mem.</td>
<td>Time</td>
</tr>
<tr>
<td>chaos</td>
<td>10s</td>
<td>64MB</td>
<td>7.4s</td>
</tr>
<tr>
<td>rayt</td>
<td>17s</td>
<td>74MB</td>
<td>14s</td>
</tr>
<tr>
<td>scimark</td>
<td>1.5m</td>
<td>16MB</td>
<td>1.4s</td>
</tr>
<tr>
<td>richards</td>
<td>10s</td>
<td>38GB</td>
<td>8.2s</td>
</tr>
<tr>
<td>unpack</td>
<td>1.1m</td>
<td>227MB</td>
<td>1.1s</td>
</tr>
<tr>
<td>go</td>
<td>30s</td>
<td>9MB</td>
<td>27s</td>
</tr>
<tr>
<td>hexiom</td>
<td>2.2m</td>
<td>1.1GB</td>
<td>1.1m</td>
</tr>
<tr>
<td>regex</td>
<td>30s</td>
<td>24MB</td>
<td>23s</td>
</tr>
<tr>
<td>process</td>
<td>14s</td>
<td>65MB</td>
<td>10s</td>
</tr>
<tr>
<td>choose</td>
<td>2.0m</td>
<td>3.2GB</td>
<td>1.1m</td>
</tr>
<tr>
<td>Total</td>
<td>6.5m</td>
<td>5.4GB</td>
<td>4.0m</td>
</tr>
</tbody>
</table>

Conclusion

- 38% analysis time improvement
- 47% less memory used
- less than 6% time spent in AGC
- no precision improvement
Conclusion
Conclusion

▶ Types or values?
Conclusion

▶ Types or values? Your choice!

Future work:
▶ Dynamic allocation policies.
▶ Scaling to more realistic Python applications.
Conclusion

- Types or values? Your choice!
- Various allocation policies, depending on the value-sensitivity.
Conclusion

- Types or values? Your choice!
- Various allocation policies, depending on the value-sensitivity.
- Abstract garbage collection is really beneficial.

Future work:
- Dynamic allocation policies.
- Scaling to more realistic Python applications.
Conclusion

▶ Types or values? Your choice!
▶ Various allocation policies, depending on the value-sensitivity.
▶ Abstract garbage collection is really beneficial.

Future work:
Conclusion

► Types or values? Your choice!
► Various allocation policies, depending on the value-sensitivity.
► Abstract garbage collection is really beneficial.

Future work:

► Dynamic allocation policies.
Conclusion

- Types or values? Your choice!
- Various allocation policies, depending on the value-sensitivity.
- Abstract garbage collection is really beneficial.

Future work:

- Dynamic allocation policies.
- Scaling to more realistic Python applications.
Thank you! Questions?

xkcd.com/353/