Value and Allocation Sensitivity in Static Python Analyses

Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné

15th June 2020

https://rmonat.fr/soap20/

Introduction

- #2 language on Github,

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`.

Based on a previous work focusing on sound type analysis[1].

[1]Monat, Ouadjaout, and Miné. "Static Type Analysis by Abstract Interpretation of Python Programs". ECOOP'20.
[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).

[1] Monat, Ouadjaout, and Miné. "Static Type Analysis by Abstract Interpretation of Python Programs". ECOOP'20.
[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.

---

[1] Monat, Ouadjaout, and Miné. "Static Type Analysis by Abstract Interpretation of Python Programs". ECOOP'20.
[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[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.
[2] Journault et al. "Combinations of reusable abstract domains for a multilingual static analyzer". VSTTE'19.

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.

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.

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

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)
```

Type Analysis

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

- 1 `ValueError`

```
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
- 1+3 `ValueError`s

```
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)
```

- 4 `ValueError`s
- l: list of `Task`s, having an integer weight

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)
```

- 4 `ValueErrors`
- l: list of `Tasks`,
  having an integer weight
- i, m: integers

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)
```

- 4 `ValueError`s
- l: list of `Task`s, having an integer weight
- i, m: integers
- `IndexError`

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)
```

- 4 `ValueError`s
- l: list of `Task`s, having an integer weight
- i, m: integers
- `IndexError`
- `NameError` (over i)
- `ZeroDivisionError`

$\Longrightarrow$ 7 false alarms

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

Refinement of the type analysis.

```
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.

```
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; $1 \leq \textit{weight} \leq 5$
- 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 `Task`s, having an integer weight; $1 \leq weight \leq 5$
- 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 `Task`s, having an integer weight; $1 \leq weight \leq 5$
- len(l)$= 4$
- $0 \leq i < 4$
- valid list access

Value Analysis

Refinement of the type analysis.

```
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 `Task`s, having an integer weight; $1 \leq weight \leq 5$
- len(l)$= 4$
- $i = 3$

Value Analysis

Refinement of the type analysis.

```
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 `Task`s, having an integer weight; $1 \leq weight \leq 5$
- len(l)$= 4$
- $i = 3$

$\Longrightarrow$ 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

Name	LOC	Type Analysis					Value Analysis				
		Time	Mem.	Exceptions detected			Time	Mem.	Exceptions detected		
				Type	Index	Key			Type	Index	Key
scimark	416	1.4s	12MB	1	1	0	3.4s	27MB	1	**0**	0
richards	426	13s	112MB	1	4	0	17s	149MB	1	**2**	0
unpack	458	8.3s	7MB	0	0	0	9.4s	6MB	0	0	0
go	461	27s	345MB	33	20	0	2.0m	1.4GB	33	20	0
hexiom	674	1.1m	525MB	0	46	3	4.7m	3.2GB	0	**21**	3
regex	1792	23s	18MB	0	2053	0	1.3m	56MB	0	**145**	0
process	1417	10s	64MB	7	7	1	12s	85MB	7	**4**	1
choose	2562	1.1m	1.6GB	12	22	7	2.9m	3.7GB	12	**13**	7
Total	9294	4.0m	2.8GB	59	2214	12	13m	9.1GB	59	228	12

Experimental Evaluation

Name	LOC	Type Analysis					Value Analysis				
		Time	Mem.	Exceptions detected			Time	Mem.	Exceptions detected		
				Type	Index	Key			Type	Index	Key
scimark	416	1.4s	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1	**0**	0
richards	426	13s	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1	**2**	0
unpack	458	8.3s	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	0	0
go	461	27s	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	33	20	0
hexiom	674	1.1m	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	**21**	3
regex	1792	23s	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	**145**	0
process	1417	10s	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	7	**4**	1
choose	2562	1.1m	[illegible]	12	22	7	2.9m	3.7GB	12	**13**	7
Total	9294	4.0m	2.8GB	59	2214	12	13m	9.1GB	59	228	12

Conclusion

The Value Analysis:

- does not remove false type alarms,
- significantly reduces index errors,
- is $\simeq 3\times$ 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
   - $(l, \boldsymbol{r})$: most recent allocation (with strong updates)
   - $(l, \boldsymbol{o})$: 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
   - $(l, \boldsymbol{r})$: most recent allocation (with strong updates)
   - $(l, \boldsymbol{o})$: older addresses (summarized)

First try: type analysis $\rightsquigarrow$
replace $l \in \mathbb{L}$ with Python types $t \in \mathbb{T}$.

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)]
```

Presenting the Recency Abstraction

$\{(\mathrm{Task}, r)$

```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:

$$\left\{ (\mathsf{Task}, r) \cdot \mathsf{weight} \mapsto [2, 2] \right.$$

```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:

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)]
```

Allocation: $(\text{Task}, r) \rightsquigarrow (\text{Task}, o)$

$$\left\{ (\text{Task}, r) \cdot \text{weight} \mapsto [2, 2] \right.$$

$$\left\{ \begin{array}{l} (\text{Task}, r) \\ (\text{Task}, o) \cdot \text{weight} \mapsto [2, 2] \end{array} \right.$$

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)]
```

$$\left\{ (\mathsf{Task}, r) \cdot \mathsf{weight} \mapsto [2, 2] \right.$$

$$\left\{ \begin{array}{l} \mathbf{(Task}, r\mathbf{)} \cdot \mathbf{weight} \mapsto \mathbf{[1, 1]} \\ (\mathsf{Task}, o) \cdot \mathsf{weight} \mapsto [2, 2] \end{array} \right.$$

Initialization:

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)]
```

Allocation: $(\mathsf{Task}, r) \rightsquigarrow (\mathsf{Task}, o)$

$$\left\{ (\mathsf{Task}, r) \cdot \mathsf{weight} \mapsto [2, 2] \right.$$

$$\left\{ \begin{array}{l} (\mathsf{Task}, r) \cdot \mathsf{weight} \mapsto [1, 1] \\ (\mathsf{Task}, o) \cdot \mathsf{weight} \mapsto [2, 2] \end{array} \right.$$

$$\left\{ \begin{array}{l} (\mathsf{Task}, r) \\ (\mathsf{Task}, o) \cdot \mathsf{weight} \mapsto [2, 2] \sqcup [1, 1] \end{array} \right.$$

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

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

$$\left\{ (\text{Task}, r) \cdot \text{weight} \mapsto [2, 2] \right.$$

$$\left\{ \begin{array}{l} (\text{Task}, r) \cdot \text{weight} \mapsto [1, 1] \\ (\text{Task}, o) \cdot \text{weight} \mapsto [2, 2] \end{array} \right.$$

$$\left\{ \begin{array}{l} \mathbf{(Task}, r\mathbf{)} \cdot \mathbf{weight} \mapsto \mathbf{[4, 4]} \\ (\text{Task}, o) \cdot \text{weight} \mapsto [1, 2] \end{array} \right.$$

Initialization:

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)]
```

Allocation: $(\text{Task}, r) \rightsquigarrow (\text{Task}, o)$

$\{(\text{Task}, r) \cdot \text{weight} \mapsto [2, 2]$

$\{(\text{Task}, r) \cdot \text{weight} \mapsto [1, 1]$
$\quad(\text{Task}, o) \cdot \text{weight} \mapsto [2, 2]$

$\{(\text{Task}, r) \cdot \text{weight} \mapsto [4, 4]$
$\quad(\text{Task}, o) \cdot \text{weight} \mapsto [1, 2]$

$\{(\text{Task}, r)$
$\quad\mathbf{(Task}, o) \cdot \mathbf{weight} \mapsto [1, 2] \sqcup [4, 4]$

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:

$$\left\{ (\text{Task}, r) \cdot \text{weight} \mapsto [2, 2] \right.$$

$$\left\{ \begin{array}{l} (\text{Task}, r) \cdot \text{weight} \mapsto [1, 1] \\ (\text{Task}, o) \cdot \text{weight} \mapsto [2, 2] \end{array} \right.$$

$$\left\{ \begin{array}{l} (\text{Task}, r) \cdot \text{weight} \mapsto [4, 4] \\ (\text{Task}, o) \cdot \text{weight} \mapsto [1, 2] \end{array} \right.$$

$$\left\{ \begin{array}{l} \mathbf{(Task}, r\mathbf{)} \cdot \mathbf{weight} \mapsto \mathbf{[5, 5]} \\ (\text{Task}, o) \cdot \text{weight} \mapsto [1, 4] \end{array} \right.$$

Variable Allocation Policies

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:

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

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

Variable Allocation Policies

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):
      ...
```

`range(L)`, `range(M)` summarized in (range, o) $\implies$ 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

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

Experimental Evaluation

Name	Type Only Partitioning			Type + Loc. Partitioning		
	Time	Mem.	⚠	Time	Mem.	⚠
chaos	**30s**	**197MB**	30	2.4m	1.2GB	**15**
rayt	27s	171MB	28	**4.5s**	**74MB**	**7**
scimark	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**3**
richards	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
unpack	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
go	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
hexiom	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
regex	[illegible]	[illegible]	[illegible]	3.6m	85MB	145
process	12s	85MB	15	**11s**	**74MB**	**13**
choose	**2.9m**	**3.7GB**	68	3.1m	4.3GB	**63**
Total	13m	9.1GB	392	87m	25GB	359

Conclusion

In most cases:

- Type-only partitioning is more efficient
- But it induces a 9% alarm overhead

Abstract Garbage Collection

Abstract Garbage Collection

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

Abstract Garbage Collection

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

```
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.

```
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.

[8] Jagannathan et al. "Single and Loving It: Must-Alias Analysis for Higher-Order Languages". POPL; Might and Shivers. "Improving flow analyses via ΓCFA: abstract garbage collection and counting". ICFP.

[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

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

Experimental Evaluation

Name	Without AGC		With AGC		Rel. Impr.	
	Time	Mem.	Time	Mem.	Time	Mem.
chaos	10s	64MB	7.4s	42MB	28%	34%
rayt	17s	74MB	14s	[illegible]	[illegible]	[illegible]%
scimark	1.5[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]%
richards	16[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]%
unpack	10[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]%
go	38[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]%
hexiom	2.2m	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]%
regex	30s	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]%
process	14s	[illegible]	10s	64MB	28%	25%
choose	2.0m	3.2GB	1.1m	1.6GB	43%	50%
Total	6.5m	5.4GB	4.0m	2.8GB	38%	47%

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!

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.

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.

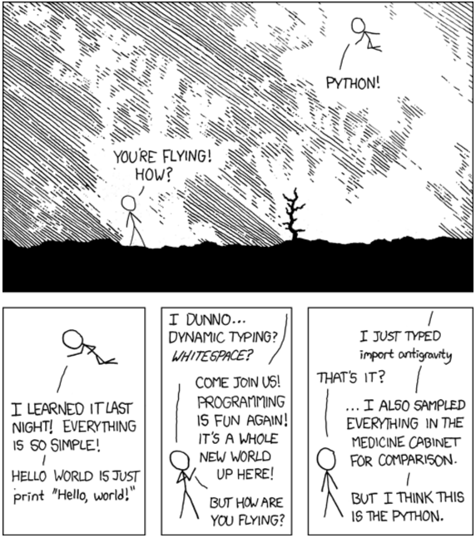
Conclusion

- 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/